**University of Alberta**

# SUBSTRING-BASED TRANSLITERATION

by

**Tarek Helmy Sherif**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 978-0-494-30022-0*
*Our file Notre référence*
*ISBN: 978-0-494-30022-0*

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

هل يمكن أن أغير مكاني من فضلك؟

*May I change my place, please?*

– Iman Mersal,
from *Al-mashy Atwal Waqt Mumkin*,
1997.

*To my family for their everlasting love and support,*
*and to Mourad and Yousef whose insanity kept me sane.*

# Abstract

Transliteration is the task of converting words, usually based on phonetics, from one writing script to another. The need for transliteration generally arises when translating a person's name, but can also occur when translating place names, organization names or borrowed words. Automating the process of transliteration is by no means a simple task, since there can be many ambiguities about the relationship between the spelling of a word and its pronunciation. Typically this task has been approached probabilistically, with probabilities assigned to different mappings between letters or phonemes, and the transliteration of a given word being selected on the basis of its likelihood. In this work, I present a novel data-driven approach to transliteration based on learning mappings between longer substrings of the two languages instead of individual letters. I show that substring-based transliteration models can outperform a state-of-the-art letter-based model.

Since the substring-based models are data-driven, I also explore means to extract training data from a bitext based on word similarity. I assess the performance of several word-similarity models on the task. I present my own method for bootstrapping a stochastic transducer and show that it outperforms the other measures on a sentence-aligned corpus.

# Table of Contents


**1 Introduction** **1**
- 1.1 Arabic-English Transliteration . . . 3
- 1.2 Machine Transliteration . . . 4

**2 Background** **8**
- 2.1 Dynamic Programming . . . 9
- 2.2 Finite-State Automata . . . 11
- 2.3 Learning Probabilities . . . 14
  - 2.3.1 The Forward-Backward Algorithm . . . 16
- 2.4 Using the Models . . . 18

**3 Previous Work** **21**
- 3.1 Transliteration Extraction . . . 21
- 3.2 Generating Transliterations . . . 24

**4 Extracting Transliterations** **28**
- 4.1 Word Similarity . . . 29
  - 4.1.1 Levenshtein Edit Distance . . . 29
  - 4.1.2 Arabic-English Fuzzy String Matching . . . 30
  - 4.1.3 ALINE . . . 32
- 4.2 Bootstrapping with a Stochastic Transducer . . . 33
- 4.3 Evaluation . . . 34
  - 4.3.1 Data . . . 34
  - 4.3.2 Experiment 1: Sentence Aligned Data . . . 35
  - 4.3.3 Experiment 2: Non-translated Data . . . 38
- 4.4 Extracting the Training Data . . . 39
- 4.5 Conclusion . . . 40

**5 Substring-Based Transliteration** **41**
- 5.1 The Noisy Channel Model . . . 42
- 5.2 Letter-based Transliteration . . . 44
  - 5.2.1 A Many-to-Many Extension to the Forward-Backward Algorithm . . . 47
- 5.3 The Monotone Search Algorithm . . . 49
- 5.4 Substring-based Transliteration . . . 50
  - 5.4.1 Viterbi Substring Decoder . . . 52
  - 5.4.2 Substring-based Transducer . . . 53
- 5.5 Experiments . . . 54
  - 5.5.1 Data . . . 54
  - 5.5.2 Evaluation Methodology . . . 55
  - 5.5.3 Setup . . . 56
  - 5.5.4 Results on the Test Set . . . 57

5.5.5 Computational Considerations . . . . . 59
5.5.6 Results on the Training Set . . . . . 60
5.5.7 Comparison to a Machine Translation System . . . . . 60
5.6 Conclusion . . . . . 62

**6 Conclusion** **63**

**A The Many-to-Many Forward-Backward Algorithm** **69**

**Glossary** **72**

# List of Tables


2.1 Dynamic programming table for the Levenshtein distance between *Samuel* and *Shimon*. . . . . 10
2.2 Adjustment of mapping and alignment probabilities over EM iterations. . . . . 15

4.1 Arabic Romanization for Levenshtein distance. . . . . 30
4.2 Letter equivalence classes for fuzzy string matching. . . . . 30
4.3 Comparison of the word-similarity models. . . . . 34
4.4 A sample of the errors made by the word-similarity metrics. . . . . 36
4.5 A sample of errors specific to each algorithm . . . . . 37
4.6 Precision of the various algorithms on the NER detection task. . . . 39
4.7 Results of bootstrapping data extraction from the remainder of the news corpus. . . . . 39

5.1 Comparison of statistical transliteration models. . . . . 54
5.2 Exact match accuracy percentage on the test set for various methods. 57
5.3 Average Levenshtein distance on the test set for various methods. . . 57
5.4 A sample of the errors made by the letter-based (LBT) and substring-based (SBT) transducers. . . . . 58
5.5 Running times and transducer sizes for a typical input word. . . . . 59
5.6 Results for testing on the transliteration training set. . . . . 60
5.7 Comparison of substring transliterators to the Google translator in terms of exact match. . . . . 60
5.8 Comparison of substring transliterators to the Google translator in terms of average Levenshtein distance. . . . . 61
5.9 A sample of the errors made by the Google translator. . . . . 61

# List of Figures


2.1 An alignment of two strings. . . . . 8
2.2 Levenshtein alignment of *Samuel* and *Shimon*. . . . . 9
2.3 Alternate alignment of *Samuel* and *Shimon*. . . . . 9
2.4 Finite-state acceptors. . . . . 11
2.5 A finite-state transducer. . . . . 12
2.6 Transducer operations. . . . . 13
2.7 A weighted finite-state transducer. . . . . 13
2.8 Enumerated alignments for EM. . . . . 15
2.9 The expectation-step function. . . . . 17
2.10 Dijkstra's shortest path search. . . . . 19

4.1 Pseudocode for the vowel normalization procedure. . . . . 31
4.2 Precision per number of words extracted for the various algorithms from a sentence-aligned bitext. . . . . 35

5.1 Examples of valid (a) and invalid (b) phrase pairs. . . . . 41
5.2 A word unigram prefix tree for the names *Najib*, *Nadia*, *Iman* and *Iran*. . . . . 45
5.3 A memoriless transliteration transducer. . . . . 46
5.4 A transliteration transducer with separate mapping probabilities for the beginning, middle and end of a word. . . . . 46
5.5 Pseudocode for a many-to-many expectation algorithm. . . . . 48
5.6 Pseudocode for a many-to-many expectation maximization algorithm. 48
5.7 High-probability (a) and low-probability (b) alignments of *Helmy* and حلمي. The Arabic is Romanized for clarity. . . . . 51
5.8 Transducers without (1) and with (2) nulls allowed in the input word. 51
5.9 The Viterbi substring decoding algorithm. . . . . 52
5.10 A one-to-one alignment of *Mourad* and مراد. For clarity the Arabic name is written left to right. . . . . 53

A.1 Pseudocode for a general many-to-many expectation maximization algorithm. . . . . 70
A.2 Pseudocode for a general many-to-many expectation algorithm. . . . 70

# Chapter 1

# Introduction

In translating a text, one often comes across words that should not or can not be translated based on meaning. Most often this is because the word refers to a named entity and thus not to the dictionary defined concept. If the two languages use the same writing script then these untranslated words can simply be transferred verbatim over to the target language. If the writing scripts differ, however, then the word is transferred in a process called **transliteration**. Transliteration is the mapping of a word from one writing script to another, usually based on the phonetics of the original word. For the purpose of this work, I will differentiate between this phonetically motivated transliteration and **Romanization**, a term I will use to refer to an orthographic mapping from a non-Latin writing script into Latin letters. In other words, Romanization is a conversion between writing scripts based on how the original word is spelled, while transliteration is a conversion based on how the original word sounds. To illustrate the difference between Romanization, translation and transliteration, consider the Arabic name نجيب محفوظ.

- Romanization: *njyb mhfwz*
- Translation: *noble preserved*
- Transliteration: *Naguib Mahfouz*

Transliteration can generally be conceived as occurring in one of two directions. **Forward transliteration** is the task of converting a word from its native script to a foreign one. *Abu Nawas*, for example, is a forward transliteration of ابو نواس to English. **Back transliteration**, on the other hand, is the restoration of a previously

transliterated name from a foreign script. If one were given the Arabic string فلادمير نابوكوف, the back transliteration would be *Vladimir Nabokov*.

One of the major difficulties in transliteration, particularly in the backward direction, is that transliteration is a **lossy** process. In other words, information is often lost about the original word when it is transliterated. There are several means by which information can be lost in transliteration.

- **Phonetic Gaps:** Different languages have different sets of available sounds. If all the sounds in a transliterated word exist in the target language, then the transliteration is a straightforward mapping from the phonemes in the source word to the letters of the target language. If, however, there are sounds missing in the target language, then these sounds must be approximated by phonemes that are available. For example, the English sound [p] does not exist in Arabic so *Paris* is transliterated as باريس [baris].

- **Pronunciation Ambiguities:** Occasionally, there is some ambiguity about how certain letters or letter combinations are to be pronounced in a given language. English, in particular, is notorious for the loose relationship between its spelling and pronunciation. The [k] sound, for example, is only represented by the letter ك in Arabic, but in English, it can be represented by *c*, *k*, *ch*, *ck* and so on. Thus, if one were given the Arabic string مايكل [mæjkal] one could not depend on phonetics alone to obtain the correct transliteration *Michael*.

- **Deleted Letters:** Related to the previous point is the fact that letter or letter combinations are occasionally deleted when transliterated. For example, the English name *Knight* is transliterated as نايت, with the silent letters simply removed.

In forward transliteration, the main objective is for the transliteration be recognizable. Thus, the pronunciation of the original word should be followed as closely as possible. In general, any phonetically reasonable transliteration is considered correct, though occasionally there is a standard transliteration (e.g. *Omar Sharif*

or *Kahlil Gibran*). Back transliteration, however, represents the more challenging of the two tasks. Information has already been lost about the word in the original transliteration, and one requires some means of recovering it. The standard of correctness is much stricter because there is generally only one correct form for a name in its native script. Phonetically sound variations are not considered acceptable. For example, محمود درويش [mahmud darwiʃ] can be acceptably transliterated as *Mahmoud Darwish*, *Mahmood Darwish* and *Mahmud Darwish*, but *Jayms Jois* is not considered an acceptable back transliteration of the Irish author's name.

## 1.1 Arabic-English Transliteration

As mentioned in the previous section, transliteration is a lossy process. The particular kind of loss that occurs depends on the pair of scripts one is dealing with. English and Arabic are a particularly interesting as languages for a study of transliteration because of the fundamental differences between them. English is a Germanic language (from the larger Indo-European family), while Arabic is a Semitic language. Both use **alphabetic scripts**, referring specifically to the fact that symbols in the two scripts are meant to represent phonemes. This differs them both from **syllabic scripts** such as Japanese katakana and **logographic** (morpheme-based) **scripts** such as Chinese hanzi. However, while English uses what is considered a **true alphabet**, meaning that both consonants and vowels are written as independent letters, the Arabic alphabet is what is known as an **abjad**, meaning that vowels are often left unwritten. In transliterating from Arabic this is obviously an issue since these unwritten Arabic vowels must be accounted for so they can be written in English.

Another major issue in Arabic-English transliteration is the vastly different phonetic inventories in each alphabet. Arabic emphatic consonants (ق,ع,ظ,ط,ض,ص,ح) are completely unavailable in English and thus must be approximated or even removed. For example, the Arabic عادل [ʔædɪl] is commonly transliterated as *Adel*. The letters خ [x] and غ [ʁ] are uncommon in English and don't generally have a standard spelling. In Arabic, glottal stops [ʔ], represented by the ء character, are

always written explicitly at the beginning of words that start with a vowel sound. English has no such restrictions on vowels written at the beginning of words. Thus, a name like *Oliver* will generally be transliterated as أوليفر, with the glottal stop added explicitly to the beginning.

From the English side, the *p* and *v* sounds do not exist in Arabic. Digraphs and silent letters are also quite common in English while they are essentially nonexistent in Arabic. The rules of pronunciation are also far more complex in English than they are in Arabic. Often, one cannot determine the pronunciation of English letters unless one is given some letter context, sometimes even requiring the entire word (e.g. the *ch* in *Michael* vs. *Richard*). This issue is compounded by the fact that pronunciation rules may differ depending on a name's language of origin. Consider the different pronunciations of the same string *Charles* in *Charles* [ʃaʁl] *Baudelaire* and *Charles* [ʧarlz] *Dickens*.

The fundamental differences between the Arabic and English alphabets present many interesting challenges in the task of transliteration in general, and thus provide an excellent test bed for attempts to automate the transliteration process.

## 1.2 Machine Transliteration

**Machine transliteration** is the task of automatically generating a transliteration for a given input word. Machine transliteration can be a useful tool for many tasks. Algorithms for machine translation and cross-language information retrieval often function by building large lexicons of translated word pairs. No matter how large these lexicons are, however, there will always be missing words. A large number of these words are generally named entities and thus candidates for transliteration. In the case of machine translation, machine transliteration can be used directly to produce a transliteration for the output text. In cross-language information retrieval, a machine transliteration model could produce candidate transliterations of named entities in a query, and these candidates could be searched for in the target text. Obviously, transliteration models are not meant to replace translation lexicons, but could be used in conjunction with a lexicon to improve performance. A lookup in a

lexicon could be used as a first step, and if the lookup fails, or some other measure deems the word a candidate for transliteration, a transliteration model could be called upon to convert the word.

In most potential applications of machine transliteration, the focus is on transliterating named entities, since, as mentioned above, named entities tend to cause the most problems for traditional lexicons. Thus, in choosing data to train and test the transliteration models presented in this work, I focused primarily on transliterations of named entities. It should be noted, however, that the algorithms presented here are sufficiently general to be used for the transliteration of other words (e.g. lexical borrowings such as تلفيزيون/*television*).

Several different approaches have been proposed to modeling transliteration, and one of the key dimensions across which these approaches differ is in how they define a basic mapping unit in a transliteration. **Phoneme-based transliteration** considers phonemes the central unit in a transliteration and thus will map the letters in different writing scripts based on the phonemes they represent. **Letter-based transliteration** considers the letters themselves to be basic units of a transliteration and thus will learn relationships between letters in the two scripts explicitly. In this work, I present a new approach to transliteration, which I refer to as **substring-based transliteration**. This approach considers the basic unit of a transliteration to be arbitrarily long substrings of letters. This looser definition of a mapping unit means a substring-based model will be able to learn longer substring mappings, allowing it to use contextual information not available to the other models.

I explore two tasks related to machine transliteration. My primary focus will be on the task of generating transliterations using my substring-based approach. The key components to a machine transliteration model are that it be accurate and generalizable. An accurate model should be able to generate desired transliterations, while a generalizable model should be able to transliterate a wide range of words. At one end of the spectrum between accuracy and generality would be a lookup table that simply stores words and their transliterations, and given one would produce the other. This model would be extremely accurate (though not necessarily perfect due to spelling variations), but no matter how large the table is, there will always be

some words missing, and these words would simply generate nothing. On the other end of the spectrum would be a simple deterministic mapping algorithm that iterates through the source string letter by letter and produces the most likely mapping for each one. This type of model would be able to generate a transliteration for any word, but it is unlikely that these transliterations would be accurate.

It is also desirable for a model to be language-independent. In other words, the model should be designed in such a way that it may be ported from one pair of languages to another with minimal modifications. For example, one could design a model for Arabic-English transliteration in which all the possible mappings between English and Arabic letters are manually encoded. Not only is this approach unattractive due to the prohibitively high number of possible mappings, but if one wanted to then port the model to Hindi-English transliteration, one would have to enumerate these mappings from scratch. Thus, state-of-the-art models of transliteration tend to be **data-driven**, meaning that the mappings are learned on the basis of examples, and the only requirement to transliterate between any given language pair is that transliteration examples for that pair be available.

Traditionally, data-driven transliteration models have centered on learning letter-to-phoneme (Knight and Graehl, 1997) or letter-to-letter (Al-Onaizan and Knight, 2002) relationships on a smaller scale. The goal was mainly to learn relationships in units that would be intuitive to a human (e.g. *sh*/ش or ʃ/ش). These models tended to borrow heavily from generative word-based models of statistical machine translation (Brown *et al.*, 1993). Recently, the trend in machine translation has been towards phrase-based models of translation (Koehn *et al.*, 2003), which is based on learning longer phrase mappings, rather than mappings between individual words. Keeping this in mind, I designed my substring-based transliteration approach as an adaptation of phrase-based translation models to the task of transliteration. My substring-based models are designed to learn mappings between longer word substrings, rather than between individual letters or phonemes. I propose two models of substring-based transliteration. The first, the **Viterbi substring decoder** uses a **dynamic programming** approach to generate transliterations. The second, the **substring-based transducer**, encodes the substring mappings into a more flexible

**finite-state transducer**.

The data-driven nature of the models discussed herein requires a large set of example transliterations from which to learn the mappings. Thus the secondary task explored in this work is the detection and extraction of transliteration pairs from a translated corpus, or bitext. Using these methods, I can create a training set of suitable size to train the generation models. I assess the performance of several measures of word similarity on the task of transliteration extraction and propose my own method which learns a word-similarity metric iteratively from the bitext itself. The transliteration pairs extracted by this method can then be used to train the transliteration models.

The remainder of this work is organized as follows: Chapter 2 presents several key concepts necessary to understanding the models presented. Chapter 3 reviews previous approaches to both generating and detecting transliterations. Chapter 4 discusses my experiments comparing different word-similarity models on the task of gathering training data for the generation models and presents my bootstrapping approach to training a stochastic transducer for the task. Chapter 5 discusses my substring-based transliteration approach in detail and presents experiments comparing it to a state-of-the-art letter-based model. Finally, Chapter 6 presents my conclusions and discusses some potential avenues for future work.

# Chapter 2

# Background

In this chapter I will discuss some of the key concepts and algorithms that recur throughout this work. The algorithms presented here function on pairs of strings which I will refer to as the **source** and **target** strings. This nomenclature comes from generative models, in which one string is assumed to produce the other, but I will retain it whether the models are generative or not. In machine transliteration for example, the source word is the given word and the target word is the transliteration produced by a model. For a given string $s$, $s_i$ will represent the $i^{th}$ letter in the string, while $s_i^{i'}$ will represent the substring beginning at position $i$ and ending at $i'$ in the string.

An **alignment** of two strings is some linking of the letters in the source and target strings (e.g. Figure 2.1). The way in which letters are linked depends on the particular task. Linked letters are referred to as **substitutions**, and if a link is between a letter and itself it can also be referred to as an **identity**. Unlinked letters in the source string are referred to as **deletions**, while those in the target string are referred to as **insertions**.

Figure 2.1: An alignment of two strings.

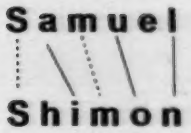


Figure 2.2: Levenshtein alignment of *Samuel* and *Shimon*.

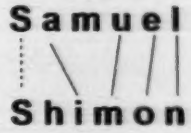


Figure 2.3: Alternate alignment of *Samuel* and *Shimon*.

## 2.1 Dynamic Programming

**Dynamic programming** refers to a class of algorithms in which the solutions to subproblems are maintained in a table to be used in finding solutions to more complex problems. As an example, consider **Levenshtein edit distance** (Levenshtein, 1966). Levenshtein distance measures the difference between two strings as the minimum number of insertions, deletions and substitutions required to convert one string into another. For example, consider the strings *Samuel* and *Shimon*. The Levenshtein distance between them is 5, and beginning with *Samuel*, the conversion could occur as follows:

1. Insert the $h \rightarrow$ *Shamuel*.
2. Substitute $i$ for $a \rightarrow$ *Shimuel*.
3. Substitute $o$ for $u \rightarrow$ *Shimoel*.
4. Delete the $e \rightarrow$ *Shimol*.
5. Substitute $n$ for $l \rightarrow$ *Shimon*.

This process can be viewed as an alignment of the two words as shown in Figure 2.2. The dashed lines represent identities and are counted as 0 in terms of the distance calculation.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| l | 6 | 5 | 5 | 5 | 5 | 5 | **5** |
| e | 5 | 4 | 4 | 4 | 4 | **4** | 4 |
| u | 4 | 3 | 3 | 3 | 3 | **3** | 4 |
| m | 3 | 2 | 2 | 2 | **2** | 3 | 4 |
| a | 2 | 1 | 1 | **2** | 3 | 4 | 5 |
| S | 1 | **0** | **1** | 2 | 3 | 4 | 5 |
| # | **0** | 1 | 2 | 3 | 4 | 5 | 6 |
| | # | S | h | i | m | o | n |

Table 2.1: Dynamic programming table for the Levenshtein distance between *Samuel* and *Shimon*.

In automating the Levenshtein distance calculation, one could exhaustively search through all possible alignments of the two words, calculating the distance for each one, and returning the minimum distance found. This is an extremely unattractive approach, however, mainly because several alignments will share links. The cost for these links will remain the same, but will be recalculated for each alignment in which they appear. Consider the alignment in Figure 2.3. The first two links are the same as above, and thus will have the exact same cost. In this exhaustive search, their cost would nevertheless be recalculated as it would be for any other alignment that begins with the same two operations.

A dynamic programming approach would be to build a table with each entry in the table representing the minimum distance for alignments of prefixes of the two words. Each row represents a letter $s_i$ in the source word, and each column, a letter $t_j$ in the target word. Each entry $(i, j)$ in the table represents the minimum cost of converting the source prefix $s_1^i$ into the target prefix $t_1^j$. The table is filled according to the following recursion.

$$\begin{aligned} Lev(0,0) &= 0 \\ Lev(i,j) &= \min \begin{cases} Lev(i-1,j)+1 \\ Lev(i,j-1)+1 \\ Lev(i-1,j-1)+sub(s_i,t_j) \end{cases} \end{aligned} \tag{2.1}$$

The function $sub(s_i, t_j)$ simply returns 0 for identities and returns 1 otherwise. The table for the Levenshtein distance calculation between *Samuel* and *Shimon* is shown in Table 2.1. The '#' symbol represents insertions or deletions occurring at

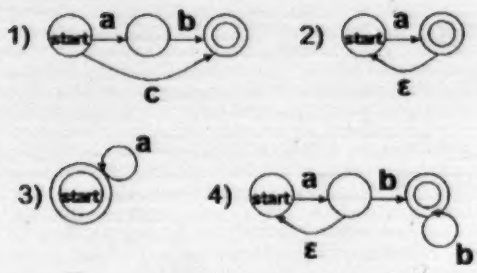


Figure 2.4: Finite-state acceptors.

the beginning of a sequence of operations.

Dynamic programming is an efficient approach to solving many problems, but it makes certain assumptions that one must be aware of when applying it to a given task. The main assumption is that the global solution can be built up in a bottom-up fashion out of the solutions to the subproblems. In other words, the optimal path taken to reach some intermediate step on the way to the global solution must be part of the path to the optimal solution. In the example presented above, the assumption is that if one found the optimal series of operations to convert the string *Samuel* into the the intermediate string *Shimoel* (step 3) then these operations must be part of the series of operations to convert *Samuel* into *Shimon*. This is known as the **dynamic programming invariant** assumption. It seems quite intuitive, but as will be discussed later in this work, there are conditions under which it must be violated.

## 2.2 Finite-State Automata

A finite-state automaton (**FSA**) is generally represented as a directed graph. It is comprised of a finite set of states (including a single **start state** and a set of **final states**) connected by directed arcs. The arcs are characterized by symbols representing the conditions under which a transition can be made from one state to another.

FSAs can be used in many applications, but I will be focusing on two: word **acceptors** and **transducers**. In a finite state acceptor, the symbols on the arcs are letters in a given alphabet, and the graph represents the set of all valid words

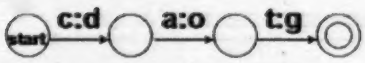


Figure 2.5: A finite-state transducer.

according to the acceptor. Given an input word the constraints in an acceptor are that transitions must begin at the start state, end at a final state, and proceed in the order that the letters occur in the word. Transitions can occur across arcs marked with the **null** ($\epsilon$) symbol without a corresponding move in the input string. Null symbols can be thought of as occurring *in between* the letters of the input string. In Figure 2.4, FSA *1* would only accept the string *ab* or the letter *c*, FSAs *2* and *3* would accept any string composed only of *a*s, and finally, FSA *4* would accept any string composed of a sequence of *a*s followed by a sequence of *b*s.

A finite-state transducer (**FST**) is also an FSA, but differs from the acceptor in that instead of defining an acceptable set of strings, it defines a set of string pairs. The only difference in terms of the graph itself is that arcs are now defined by pairs of symbols. Although FSTs can be used as acceptors for pairs of strings, in this work I will be focusing on their function as a generators. In other words, an FST will be given one string as input (with symbols matching one side of the symbol pairs on the arcs) and will produce a corresponding output string. Transducers are built to function in both directions, so the input could be made to either the left or right set of symbols on the arcs, with the output corresponding to the opposite side. For example, the transducer in Figure 2.5 could take the string *cat* as input to the left side and would then generate the string *dog*. It could analogously take the string *dog* as input to the right side, and would then generate *cat*.

Finite-state transducers are closed under the following three operations (meaning that the result will still be an FST):

- **Inversion:** a reversal of the left and right sides of the symbol pairs on the arcs. In other words, the input to the left of the inverted transducer will produce the same output as right input to the original transducer.

- **Union:** the union of two transducers can simply take all the same input as the

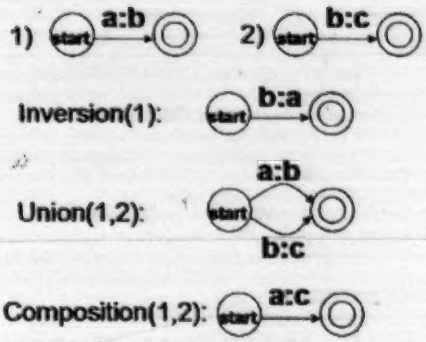


Figure 2.6: Transducer operations.

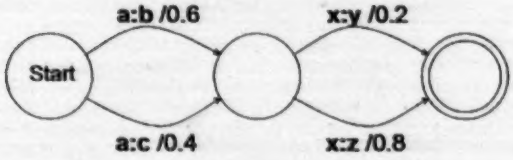


Figure 2.7: A weighted finite-state transducer.

original transducers and will produce the same output.

- **Composition:** the output from one transducer is used as input to the next. More formally, if FST $a$ maps input strings $I_a$ to output strings $O_a$, and FST $b$ maps from $I_b$ to $O_b$, then the composition of $a$ and $b$ would map from $I_a$ to $O_b$.

Figure 2.6 presents examples of these operations.

The automata presented so far in this section show no preference to one string or pair of strings over another. Often, however, particularly in the case of transducers where a single input string can produce multiple output strings, it is desirable to rank paths through the automaton in some way. In a statistical approach, this is done by assigning probabilities to the arcs, and the probability of a given path through the automaton is simply the product of the probabilities of its individual arcs. This augmented automaton is referred to as a weighted finite-state automaton (**WFSA**). For example, if the string $ax$ were input into the left side of weighted finite-state transducer (**WFST**) shown in Figure 2.7, there are four potential outputs: *by* with a

probability of 0.12, *bz* with a probability of 0.48, *cy* with a probability of 0.08 and *cz* with a probability of 0.32. Thus, the outputs can be ranked according to their probabilities, and depending on the task, a specific output can be selected.

## 2.3 Learning Probabilities

In designing a probabilistic model, one of the main issues that arises is how to assign probabilities to given events. In the case of transliteration, one would want to assign probabilities to individual mapping operations, such as *d*/ɔ. One could attempt to assign these probabilities manually, based on intuitions about the languages being transliterated, but this approach is unattractive for two reasons. First of all, this would require enumerating all possible mappings by hand, which, as mentioned in the previous chapter, is tedious and generally unfeasible. The other drawback is that it would be difficult to ground these probabilities in anything concrete. Should the probability of a *d*/ɔ mapping be 0.6 or 0.8 or 0.999? There is no concrete way to justify one probability over another.

**Expectation maximization (EM)** (Baum, 1972) is a statistical approach to learning these probabilities from a set of training examples. If the task were to learn the probabilities for letter mappings between words, an EM algorithm would begin with a random guess as to what the individual mapping probabilities are. It would then iteratively adjust the mapping probabilities based on how the mappings appear in the data. Given a set of example word pairs the process could be outlined as follows:

1. Assign an equal probability to all mapping operations.
2. Enumerate all possible alignments for each pair of words.
3. Calculate the probability of an alignment as the product of the probabilities of its individual mappings.
4. Normalize the alignment probabilities so that they sum to 1 for each pair of words.

| Iteration | Mappings | | | | | Alignments | | |
|---|---|---|---|---|---|---|---|---|
| | P(a,x) | P(ϵ,y) | P(a,y) | P(ϵ,x) | P(b,z) | P(1a) | P(lb) | P(2) |
| 1 | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 | 0.5 | 0.5 | 1.0 |
| 2 | 0.38 | 0.13 | 0.13 | 0.13 | 0.25 | 0.75 | 0.25 | 1.0 |
| 3 | 0.44 | 0.19 | 0.06 | 0.06 | 0.25 | 0.96 | 0.04 | 1.0 |
| 4 | 0.49 | 0.24 | 0.01 | 0.01 | 0.25 | 0.999 | 0.001 | 1.0 |

Table 2.2: Adjustment of mapping and alignment probabilities over EM iterations.

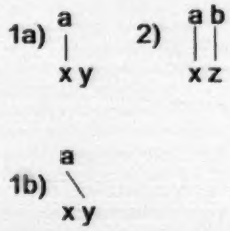


Figure 2.8: Enumerated alignments for EM.

5. Collect **partial counts** for each mapping operation in each alignment. The partial count collected from a particular alignment is simply the count weighted by the probability of that alignment.

6. Normalize the mapping probabilities so that they sum to 1.

7. Repeat steps 3-6 until the probabilities cease to change.

For example, assume one is given two training examples: *a/xy* and *ab/xz*. For simplicity the alignments will be constrained so that each letter in the source string must be linked to exactly one letter in the target string, and each letter in the target string can link to at most one letter in the source string. The possible alignments are enumerated in Figure 2.8. Due to the aforementioned constraint there is no ambiguity about the alignment of the second pair (alignment 2), but the first pair can be aligned in two ways, and the algorithm must choose between them if it is to learn accurate mappings. Intuitively, alignment 1*a* is more attractive since the *a/x* link agrees with alignment 2. Table 2.2 shows how the probabilities of the mappings and alignments change over the course of EM iterations. The probabilities shown are as they would be after the normalization of the alignment probabilities (step 4

above). It is clear that as the iterations progress, the algorithm is leaning towards the more intuitively preferable alignments and is assigning the mapping probabilities accordingly.

### 2.3.1 The Forward-Backward Algorithm

In the EM example described above, the constraint on letter-to-letter mappings restricted the number of alignments that had to be enumerated. In a real application, however, constraints are likely to be much looser. This would lead to a similar problems as with Levenshtein distance (Section 2.1) in that the probabilities for several subsections of the alignments have to be calculated repeatedly. The solution, once again, is to apply dynamic programming.

The **forward-backward algorithm** uses two tables to store probabilities for portions of an alignment and uses these values to calculate alignment probabilities and collect partial counts for the mapping operations. Ristad and Yianilos (1998) present a forward-backward algorithm for training a one-state weighted transducer.

The forward table $F$ is similar to the Levenshtein table described in Section 2.1 in that each entry $(i, j)$ represents alignments of $s_1^i$ and $t_1^j$. The operations allowed are the same as before as well: insertions, deletions or substitutions. In this probabilistic setting, however, the value stored in each entry is the sum of the probabilities of all possible alignments of $s_1^i$ and $t_1^j$. The backward table $B$ analogously stores, at each $(i, j)$ entry, the sum of the probabilities of all alignments of the *suffixes* $s_{i+1}^I$ and $t_{j+1}^J$, where $I$ and $J$ represent the lengths of source string $s$ and target string $t$, respectively. The tables are filled according to the following recursions.

$$\begin{aligned} F(0,0) &= 1 \\ F(i,j) &= P(s_i,\epsilon)F(i-1,j) \\ &\quad +P(\epsilon,t_j)F(i,j-1) \\ &\quad +P(s_i,t_j)F(i-1,j-1) \end{aligned} \tag{2.2}$$

**Algorithm** expectation-step $(s, t)$

**for** $i = 0...I$
  **for** $j = 0...J$
    **if** $(i > 0)$
      $C(s_i, \epsilon) += F(i-1, j)P(s_i, \epsilon)B(i, j))/F(I, J)$
    **if** $(j > 0)$
      $C(\epsilon, t_j) += F(i, j-1)P(\epsilon, t_j)B(i, j))/F(I, J)$
    **if** $(i > 0 \wedge j > 0)$
      $C(s_i, t_j) += F(i-1, j-1)P(s_i, t_j)B(i, j))/F(I, J)$

Figure 2.9: The expectation-step function.

$$\begin{aligned} B(I, J) &= 1 \\ B(i, j) &= P(s_{i+1}, \epsilon)B(i+1, j) \\ &\quad + P(\epsilon, t_{j+1})B(i, j+1) \\ &\quad + P(s_{i+1}, t_{j+1})B(i+1, j+1) \end{aligned} \tag{2.3}$$

Once the $F$ and $B$ tables are filled, the *expectation-step* function (Figure 5.5) can be called to collect the partial counts which are stored in the $C$ table. Recall that $F(i, j)$ stores the probability of all possible alignments of $s_1^i$ and $t_1^j$, while $B(i, j)$ contains the probabilities for all alignments of $s_{i+1}^I$ and $t_{j+1}^J$. Thus, the substitution count $C(s_i, t_j)$ collected at a particular $i$ and $j$ will be for all alignments that link $s_i$ with $t_j$. Counts for insertions and deletions are collected similarly. Once the counts are collected they can be normalized to create a probability distribution.

One can see that the above algorithm functions equivalently to the exhaustive EM approach described earlier in this section. The filling of the $F$ and $B$ tables carries out steps 2 and 3, while the division by $F(I, J)$ in the *expectation-step* function performs step 4 (recall that $F(I, J)$ contains the sum of the probabilities of all alignments of the two words). Finally, step 5 is performed by the *expectation-step* function with the only difference being that counts are collected from several alignments simultaneously. All other steps are performed exactly as in the original approach.

## 2.4 Using the Models

Once the structure of a statistical model has been defined, and the probabilities have been learned, one needs a method to extract information from the model. This can be referred to as searching through or **decoding** a model. The type of decoding done depends on the information desired from the model. For example, if the desired information is a score for similarity between two strings, the forward algorithm (Equation 2.2) can be used. In this case the similarity score is defined as the sum of the probabilities of all paths through a transducer that correspond to the two words. After running the algorithm, this value would be stored in $F(I, J)$.

On the other hand, there are occasions when only the single most probable path is required, for example when creating an alignment between two strings. The forward algorithm would be inappropriate in this case. This is because it considers all paths simultaneously, and at the same time is making the dynamic programming invariant assumption. Thus, it is impossible to extract a single path from the table. The **Viterbi algorithm** (Viterbi, 1967) is a dynamic programming method that differs from the forward algorithm in that it only considers the single most probable path through a model. For a source string $s$ of length $I$ and target string $t$ of length $J$, a table $V$ is filled according to the following recursion:

$$\begin{aligned} V(0,0) &= 1 \\ V(i,j) &= \max \begin{cases} P(s_i, \epsilon) V(i-1, j) \\ P(\epsilon, t_j) V(i, j-1) \\ P(s_i, t_j) V(i-1, j-1) \end{cases} \end{aligned} \tag{2.4}$$

After running the algorithm, the probability of the most probable path through the model will be stored at $V(I, J)$. If the alignment itself is desired, back pointers can be kept to maximizing arguments to recreate the path.

The dynamic programming implementation of the Viterbi algorithm makes certain assumptions about the state space it is searching through. The main assumption is that states are defined by a source-target letter pair $(s_i, t_j)$. If a more flexible state space is being used, another option is **Dijkstra's shortest path search** (Dijkstra, 1959). The algorithm makes similar assumptions to dynamic programming algo-

**Algorithm** Dijkstra-decode $(WFST, t, start, final)$

**foreach** state $q$ **in** $WFST$ **do**
  $p(q) := 0$
  $previousq := \epsilon$
  $known(q) := false$
$p(start) := 1$
$known(start) := true$
$PQ.add(start)$
**while** $\neg PQ.isEmpty()$ **do**
  $q := PQ.getMax()$
  **if** $q = goal$ **[end loop]**
  $A := A \cup q$
  **foreach** arc $(q, q')$ outgoing from $q$ **do**
  **if** $p(q) * t(q, q') > p(q')$
    $p(q') := p(q) * t(q, q')$
    $previous(q') := q$
    **if** $\neg known(q')$
      $PQ.add(q')$
      $known(q') := true$

Figure 2.10: Dijkstra's shortest path search.

rithms, but does not require that the state space be defined by letters in the source and target strings. The search essentially functions by always taking the shortest (most probable, in a statistical setting) arc outward from the periphery of the search and storing the shortest known path to any state seen thus far. A probabilistic version of the algorithm for a weighted FST is outlined in Figure 2.10. $PQ$ is a priority queue that simply returns the state with the highest $p(q)$ score. Note that the $p(q)$ score is not the probability of state $q$ or the transition leading to $q$ but the probability of the entire path leading up to and including state $q$. The transition table $t(q, q')$ simply stores the probabilities on the arcs between states.

# Chapter 3

# Previous Work

Several approaches have been proposed for tasks related to transliteration. In this chapter, I will review some of the literature related to the two main tasks discussed in this work: extraction and generation of transliterations. In extracting transliterations, one is given two words and must decide whether or not they are transliterations of each other. Generating transliterations is a much more difficult task. One is simply given a word in one writing script and must produce the transliteration in another writing script. Since only one word is given, there is less information available than in the detection task.

## 3.1 Transliteration Extraction

Traditional approaches to building translation lexicons have been based on models of machine translation (Brown *et al.*, 1993), with refinements such as those presented in (Melamed, 1996). These approaches tend to focus on finding words that occur with similar frequencies and tend to co-occur. Transliterated words, however, tend to occur infrequently in any given bitext, and thus it is hard to make any judgements about their co-occurrence using these traditional methods.

Collier *et al.* (1997) propose a method for detecting transliterations between English and Japanese katakana in a bitext. Their model first transcribes the katakana word as a single intermediate representation of all possible transliterations of the individual symbols. For example if katakana symbol *x* could be transliterated as *bu* or *bo*, and katakana symbol *y* as *ru*, *ro*, *lu* or *lou*, the resulting representation

of *xy* in English would be *bourlou*. A depth-first search is then used to count the number of matching letters between the intermediate representation and a candidate English transliteration. For each katakana sequence in a Japanese article, the metric was used to find a matching English proper noun in the article's English translation.

Tsuji (2002) proposes a method for building transliteration rules manually between katakana and English. The katakana strings are split into their mora units, and the English mappings of each unit are then assessed manually from a given set of training pairs. Mapping rules for each mora unit are ranked according to their frequency. For each katakana string in a bitext, all possible transliterations are generated based on the mapping rules. These possible transliterations are then compared to all English words in the corresponding English translations (the bitext consisted of translated article titles and abstracts from journals in various fields). The transliteration candidates are ranked according to the Dice score which measures similarity by comparing the length of the longest subsequence shared by the two words to the lengths of the two words. This method is computationally expensive, since all possible transliterations of the katakana word must be compared to all English words in the translations. To remedy this, a heuristic is suggested in which longer words are allowed less mapping rules per mora unit. In reducing the list of allowable mappings for a mora unit, selections are made according to the frequencies assessed from the training set.

Lee and Chang (2003) use a generative noisy channel transliteration model, similar to the transducer presented in (Knight and Graehl, 1997), to extract English-Chinese transliterations. EM is used to learn the mapping probabilities based on a many-to-many Viterbi alignment of English and Chinese symbols defined by the following recursion:

$$\begin{aligned} V(0,0) &= 1 \\ V(i,j) &= \max_{h,k} P(C_{j-k}^{j} | E_{i-h}^{i}) V(i-h, j-k) P(h,k) \end{aligned} \tag{3.1}$$

$P(h,k)$ represents the probability of a mapping occurring between an English sequence of length $h$ and a Chinese sequence of length $k$. A key difference between the EM training proposed here and the one presented in (Knight and Graehl, 1997) is the fact that this algorithm only considers the single most probable alignment

(and not all possible alignments) when calculating mapping probabilities. To extract Chinese transliterations, the English side of the bitext is first tagged with a named entity tagger. The model is then used to isolate the transliteration in the Chinese translation.

Klementiev and Roth (2006) propose a model to extract Russian transliterations of English named entities from comparable Russian-English news corpora. The term comparable corpora is used to describe pairs of Russian and English news articles that can be considered loose translations of each other (e.g. describing the same event). They use a bootstrapping approach to train a perceptron as a discriminative transliteration model, and employ Fourier analysis to compare distributions of words over time. The perceptron functions by splitting the English and Russian words into all their constituent n-grams. English-Russian n-gram pairs are then used as features for the perceptron. For example, if the n-grams were of size 2, and the strings being compared were *abc* and *xyz*, the features for the perceptron would be $((a, x), (ab, x), (ab, xy), ...(bc, yz), (c, yz), (c, z))$. When training begins, the perceptron is trained on a small seed set of known transliterations. The English side of the corpus is tagged by a named entity tagger, and the perceptron proposes transliterations for the named entities. The candidate transliteration pairs are then reranked according to the similarity of their distributions across dates, and candidates scoring above a certain threshold are used to train the perceptron for the next iteration. These steps are repeated until the training set ceases to change, and then the top scoring candidate for each English named entity, according to the perceptron and the Fourier analysis, is selected.

Freeman *et al.* (2006) propose an Arabic-English fuzzy matching algorithm to extract transliterations from a bitext. The model is built with Levenshtein edit distance (Chapter 2) as its base, but with the additional encoding of a great deal of knowledge about the relationships between English and Arabic letters. Equivalence classes, which are given a substitution cost of 0, are created between English and Arabic letters. Several rule-based transformations are also performed on word pairs before they are compared. This model will be discussed in greater detail in Chapter 4.

## 3.2 Generating Transliterations

Arbabi *et al.* (1994) propose to model forward transliteration through a combination of neural net and knowledge-based systems. They suggest that the main difficulty in transliterating Arabic names is that short vowels are rarely written in Arabic text, and thus their main task was to vowelize the Arabic names as a preprocessing step for transliteration. The knowledge-based system (KBS) is a set of prioritized *IF-THEN* rules. The majority of the rules are for vowelization based on patterns in the input word, but there are also rules to filter out corrupt data and to perform table lookups. The KBS's first task is to filter out corrupt data from the input and to find names that can be vowelized using a simple lookup table. The remaining names are then passed to an artificial neural network (ANN). The ANN is trained on names from an Arabic phone-book and is meant to discriminate between *reliable* and *unreliable* names. The reliability of a name is defined on the basis of the KBS's ability to vowelize it correctly. The reliable names are then passed back to the KBS to be vowelized based on the vowelization rules therein. There are two major concerns with this approach. The first is that it is Arabic-specific. Not only are the rules specific to Arabic-English transliteration, and thus would require a complete overhaul if they were to be used on other languages, but it deals with the very specific issue of vowelization which is not a concern for all language pairs. The second issue is that the requirement of reliability severely limits its applicability even for Arabic-English transliteration.

Knight and Graehl (1997) propose to statistically model the transliteration of Japanese syllabic *katakana* script into English. They use the noisy channel approach (Brown *et al.*, 1990) to describe an English name being transformed into katakana according to the following generative process:

1. An English name $E$ is generated with probability $P(E)$.

2. The English name $E$ is converted into phonetic sequence $E'$ with probability $P(E'|E)$.

3. The English phonetic sequence $E'$ is converted into a Japanese phonetic se-

quence $J'$ with probability $P(J'|E')$.

4. The Japanese phonetic sequence $J'$ is converted into a katakana string $J$ with probability $P(J|J')$.

5. Spelling errors are introduced into the katakana string $J$ with probability $P(O|J)$.

$P(E)$ is learned based on counts from a large English corpus. $P(E'|E)$ is learned from the CMU pronunciation dictionary. $P(J'|E')$ is learned using the EM algorithm which was discussed in detail in Chapter 2. $P(J|J')$ is defined manually. This is possible because katakana symbols tend to be very regular in their pronunciation. Finally, $P(O|J)$ is learned by EM using examples of correctly spelled katakana names and the same names corrupted by an optical character recognizer.

Once the probabilities are learned, the task can be defined as follows. Given a (possibly misspelled) katakana string $O$, one wishes to find an English name $\hat{E}$ that maximizes the product of the above probabilities. More formally, we want $\hat{E}$ where

$$\hat{E} = \arg\max_{E} P(O|J)P(J|J')P(J'|E')P(E'|E)P(E)$$

To do so, each of the probabilities are encoded as a separate weighted finite-state transducer, and the sequence of transducers is composed. Since, as discussed in Chapter 2, WFSTs can take input from either side, the fact that the transliteration is being modeled from English to katakana is inconsequential. The katakana string can simply be input to the katakana side, with the resulting output being the transliteration proposed by the transducer. Stalls and Knight (1998) adapt this approach to Arabic, with the modification that the English phonemes are mapped directly to Arabic letters. This was due to the fact that Arabic symbols are not as regular in their pronunciation as katakana symbols and that pronunciation dictionaries for Arabic are uncommon. Al-Onaizan and Knight (2002) find that a transducer mapping directly from English to Arabic letters outperforms the phoneme-to-letter model. Further discussion of these transducer-based approaches can be found in Chapter 5.

AbdulJaleel and Larkey (2003) propose a model for forward transliteration from Arabic to English that also uses the noisy channel approach. Given an Arabic word $A$, they wish to find the English word $\hat{E}$ such that

$$\hat{E} = \arg\max_{E} P(A|E)P(E)$$

Note that the probability $P(A|E)$ indicates mapping probabilities are learned between Arabic and English strings directly, without any phonetic conversion. Instead of using EM directly to learn the mapping probabilities, they use a statistical word alignment model, GIZA++ (Och and Ney, 2000), to align the letters in English-Arabic word pairs. This was done by simply splitting the words into individual letters and considering the letters as words in the alignment model. The alignment model is used in two stages. First, an alignment is found between the individual letters in each word pair. This alignment is used to extract English n-grams, based on cases where the alignment links more than one English letter to one Arabic letter. The top 50 most frequent n-grams are considered single symbols for the second stage of alignment. In the second stage, the English words are resegmented to include the n-grams as individual symbols and the words are realigned. Counts are collected for the individual mappings in each alignment and these counts are used to create a conditional probability distribution for Arabic symbols given English symbols. $P(E)$ is defined as letter-bigram model. Thus, the two probabilities in the equation described above can be defined as

$$P(A|E) = \prod_{i} P(A_i|E_i)$$

and

$$P(E) = \prod_{i} P(E_i|E_{i-1})$$

Li *et al.* (2004) argue that traditional generative approaches to machine transliteration do not encode enough contextual information, and that this contextual information is necessary for Chinese-English transliteration. They propose to model transliteration as joint process. In other words, instead of assuming that one word generates the other ($P(E|C)$), the two words are assumed to be generated simultaneously ($P(E, C)$) by some underlying hidden process. Their model defines this

joint process as n-grams of mapping units. Thus, a transliteration pair $(E, C)$ is broken down into mapping units $(< E_1, C_1 >, < E_2, C_2 >, ...)$, where $E_i$ and $C_i$ can consist of one or more characters in their respective languages. N-grams can then be learned over these mapping units. For example, in the case of a bigram model, the model would define the probability of a transliteration pair as

$$P(E, C) = \prod_i P(< E_i, C_i > | < E_{i-1}, C_{i-1} >)$$

The model learns the n-gram probabilities according to the following EM algorithm:

1. Create initial random alignment.
2. Update n-gram statistics (of mapping units) to estimate probability distribution.
3. Apply new n-gram probability model to obtain new alignment.
4. Repeat steps 2-3 until alignment converges.
5. Derive a list of potential transliteration units from the final alignment.

Once these probabilities are learned, the task is to find, given a Chinese word $C$, an English word $\hat{E}$ such that

$$\hat{E} = \arg\max_E P(E, C)$$

Ekbal *et al.* (2006) adapt this model to the transliteration of names from Bengali to English, with the modification that mapping units are not discovered through an alignment process. Knowledge about the units that tend to be transliterated from Bengali to English is encoded into regular expressions which are used to extract the mapping units.

# Chapter 4

# Extracting Transliterations

The generation models that will be presented in Chapter 5 are data-driven and thus require a reasonable number of sample transliterations to learn from. These examples could, of course, be acquired manually, but this would be an extremely tedious task. Thus, I was led to explore means to extracting sample transliterations automatically from a bitext. Many of the difficulties involved in transliteration detection are general issues related to the task of transliteration itself (deletions, pronunciation ambiguities, etc. see Chapter 1). There are some problems, however, that are unique to detection. In general, since one is simply given a translated corpus, there is no guarantee that either of the words being compared is meant to be transliterated. This poses a problem in particular with names that also have a meaning as words in a given language. For example, the word *Brown* could refer to a person's name or the colour, and the word كنت could be a transliteration of the name *Kent* or it could mean "*I was*." A related problem is that of names being matched to unrelated words in the translated text. For example, the English *Mars* (the planet) could, by many measures of similarity, be mistaken for a transliteration of the Arabic مارس [mærasa] (meaning "*he practiced*"). If the two happened to appear in a translated sentence pair, they could be incorrectly marked as transliterations.

In this chapter, I will be exploring the use of several word-similarity metrics to discover transliterations in a bitext. A word-similarity metric is any measure of the similarity between two strings. Levenshtein edit distance, presented in Chapter 2, is an example of such a metric. In general, word-similarity metrics depend on the strings being written in the same script. Levenshtein distance, for example, must

find identities to be able to give a meaningful measure. Since Arabic and English use different writing scripts, one must determine how the letters in each language relate to each other. One option is to map the strings to some common script, be it orthographic or phonetic. Another is to manually identify the similarity between the letters of the two scripts.

An issue with the solutions presented above is that they depend, to greater or lesser degrees, on knowledge of the languages being compared. With this in mind, I present a bootstrapping approach to training a weighted transducer. Bootstrapping refers to training methods that begin with a small set of labeled training examples (in this case, known transliterations) and uses models trained on these to label unknown examples and incorporate them into the training set iteratively. This method can function on any pair of writing scripts without any required preprocessing.

## 4.1 Word Similarity

In this section, I present three models of word similarity. A key dimension across which these algorithms differ is their specificity to any particular language. Language-specific models are designed for a particular language or pair of languages and are not easily ported to others, while language-independent models are designed without this limitation. As some of them require that the word pairs being evaluated be written in the same alphabet, there is also a difference in the amount of preprocessing required before evaluation. All three models use hard-coded values, meaning that the values of individual mapping units used in the comparison are determined a priori.

### 4.1.1 Levenshtein Edit Distance

As a baseline for my experiments, I used Levenshtein edit distance, which was discussed in Chapter 2. The algorithm simply counts the minimum number of insertions, deletions and substitutions required to convert one string into another. Levenshtein distance is essentially language-independent, but since the measure depends on finding identical letters, both words must use the same alphabet. Prior to

| ا,أ,ة,آ,ء → a | ب → b | ت,ط → t | ث,ذ,ظ → th |
|---|---|---|---|
| ج → j | ح,ه → h | خ → kh | د,ض → d |
| ر → r | ز → z | س,ص → s | ش → sh |
| ع → ' | غ → g | ف → f | ق → q |
| ك → k | ل → l | م → m | ن → n |
| و → w | ي → y | | |

Table 4.1: Arabic Romanization for Levenshtein distance.

| ا,أ,آ,ى ↔ a,e,i,o,u | ب ↔ b,p,v | ت,ط,ث ↔ t | ج ↔ j,g |
|---|---|---|---|
| ح,ه ↔ h | خ ↔ k | د,ض,ذ ↔ d | ر ↔ r |
| ز ↔ z | س ↔ s,c | ص,ش ↔ s | ظ ↔ d,z |
| ع,ء ↔ ',c,a,e,i,o,u | غ ↔ g | ف ↔ f,v | ق ↔ q,g,k |
| ك ↔ k,c,s | ل ↔ l | م ↔ m | ن ↔ n |
| و ↔ w,u,o | ي ↔ y,i,e,j | ة ↔ a,e | |

Table 4.2: Letter equivalence classes for fuzzy string matching.

comparison, the Arabic words are Romanized based on intuitive mappings for each letter (Table 4.1). The distances are also normalized by the length of the longer of the two words to avoid excessively penalizing longer words.

### 4.1.2 Arabic-English Fuzzy String Matching

The fuzzy string matching algorithm proposed in (Freeman *et al.*, 2006) uses Levenshtein distance as its base, but with the addition of a great deal of knowledge about the relationships between English and Arabic letters. The first addition is that of equivalence classes between English and Arabic letters (Table 4.2), meaning that certain Arabic and English letters can be matched to each other with 0 cost. For example, the Arabic ف can match both *f* and *v* in English with no cost. Another major modification is the normalization of the candidate word pair in two ways. The first is to perform a rule-based letter normalization of both words. Some examples of normalization include:

- English double letter collapse: e.g. *Miller*→*Miler*.

**Algorithm** VowelNorm ($Estring, Astring$)

**for each** $i := 0$ **to** $min(|Estring|, |Astring|)$
  **for each** $j := 0$ **to** $min(|Estring|, |Astring|)$
    **if** $Astring_i = Estring_j$
      $Outstring. = Estring_j; i++; j++;$
    **if** $vowel(Astring_i) \wedge vowel(Estring_j)$
      $Outstring. = Estring_j; i++; j++;$
    **if** $\neg vowel(Astring_i) \wedge vowel(Estring_j)$
      $j++;$
      **if** $j < |Estring_j|$
        $Outstring. = Estring_j; i++; j++;$
    **else**
      $Outstring. = Estring_j; i++; j++;$
**while** $j < |Estring|$
  **if** $\neg vowel(Estring_j)$
    $Outstring. = Estring_j;$
  $j++;$
**return** $Outstring;$

Figure 4.1: Pseudocode for the vowel normalization procedure.

- Arabic hamza collapse: e.g. أشرف→اشرف.
- Individual letter normalizations: e.g. *Hendrix*→*Hendriks* or شريف→سهريف.

The second is to iterate through both words and remove any vowels in the English word for which there is no similarly positioned vowel in the Arabic word. The pseudocode for my implementation of this vowel normalization is presented in Figure 4.1. Freeman *et al.* (2006) also proposed two modifications for special cases, stemming affixed Arabic prepositions and a second pass to handle the ambiguous English *ch*, but they reported that these modifications had a minimal effect on performance. Thus, the equivalence classes and normalizations are the only modifications I reproduce for my experiments here.

After letter and vowel normalization, the standard Levenshtein algorithm is run using the letter equivalences as matches instead of identities. The distance is normalized by the sum of the lengths of both words. Unlike the other metrics presented in this section, this algorithm can only function on Arabic and English.

### 4.1.3 ALINE

The ALINE algorithm (Kondrak, 2000) differs from the other algorithms presented here, mainly in that it functions on phonetic transcriptions of the words being compared, instead of the orthographic forms. It was originally designed to identify cognates in related languages, but as long as the word pairs are transcribed using phonemes available to the algorithm, it can be used to compare any pair of words.

Individual phonemes input to the algorithm are expressed as around a dozen phonetic features, such as Place, Manner and Voice. Each feature is given a value between 0 and 1, and each is weighted according to its relative importance. The values of each feature represent the distance between vocal organs during speech production.

The core function in ALINE compares a pair of phonemes and assigns a score based on the similarity as assessed by a comparison of the individual features. An optimal alignment of the two words is computed with a dynamic programming algorithm (Wagner and Fischer, 1974), and the overall score is the sum of the scores

of all links in the alignment. Insertion and deletion penalties are constant, and vowel matches are assigned less weight in the scoring than are consonant matches.

In my experiments, the Arabic and English words are converted into phonetic transcriptions using a deterministic rule-based transformation. English vowels are not converted as their conversion is not trivial, and Arabic emphatic consonants are depharyngealized. After comparison, the score for a pair of words is normalized by the length of the longer of the two words.

## 4.2 Bootstrapping with a Stochastic Transducer

The probabilistic model for string similarity outlined in (Ristad and Yianilos, 1998) was discussed in Chapter 2. EM training is used to learn probabilities for a memoriless transducer, which is then used to assign probabilities to pairs of words based on their similarity.

The **forward algorithm**, which is part of the forward-backward algorithm used to train the model, can also be used to score a pair of words. It is reprinted here for convenience.

$$\begin{aligned} F(0,0) &= 1 \\ F(i,j) &= P(s_i,\epsilon)F(i-1,j) \\ &\quad +P(\epsilon,t_j)F(i,j-1) \\ &\quad +P(s_i,t_j)F(i-1,j-1) \end{aligned} \tag{4.1}$$

Once the algorithm is run, the value stored at $F(I,J)$ is the sum of the probabilities of all paths through the transducer that correspond to the two words being compared.

The major issue in porting the memoriless transducer over to the task of transliteration detection is that the training, as outlined in (Ristad and Yianilos, 1998), is supervised. In other words, it would require a relatively large set of known transliterations for training, and this is exactly what I am looking to use the model to acquire. To overcome this problem, I look to the **bootstrapping** method outlined in (Yarowsky, 1995). Yarowsky trained a rule-based classifier for word sense disambiguation by starting with a small set of seed examples for which the sense was

| | Levenshtein | Fuzzy Match | ALINE | Bootstrap |
|---|---|---|---|---|
| Lang.-specific | No | Yes | No | No |
| Preprocessing | Romanization | None | Phon. Conversion | None |
| Learned | No | No | No | Yes |

Table 4.3: Comparison of the word-similarity models.

known. The trained classifier was then used to label examples for which the sense was unknown, and these newly labeled examples were then used to retrain the classifier. These steps were repeated until convergence.

My method uses a similar approach to train a stochastic transducer. The algorithm proceeds as follows:

1. Add seed pairs to the training set.
2. Train the transducer using the forward-backward algorithm on the current training set.
3. Calculate the forward score for all word pairs under consideration.
4. If the forward score for a pair of words is above a predetermined threshold, add the pair to the training set.
5. Repeat steps 2-4 until the training set ceases to grow.

Once training stops, the transducer can be used to score pairs of words not in the training set. For my experiments, the scores were normalized by the average of the lengths of the two words. A comparison of the four models discussed is presented in Table 4.3.

## 4.3 Evaluation

### 4.3.1 Data

The two bitexts used for this task were the Arabic Treebank Part 1-10k word English Translation corpus and the Arabic English Parallel News Part 1 corpus (approx. 2.5M words). Both bitexts contain Arabic news articles and their English

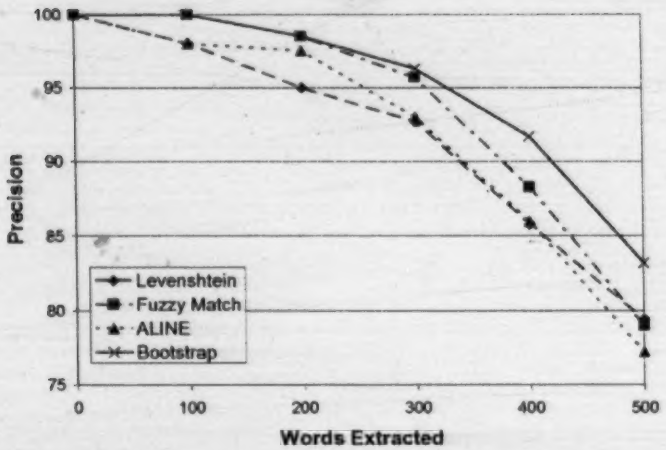


Figure 4.2: Precision per number of words extracted for the various algorithms from a sentence-aligned bitext.

translations aligned at the sentence level and both are available from the Linguistic Data Consortium. The Treebank data was used as a development set, and testing was done on the first 20k lines (approx. 50k words) of the parallel news data.

The data was preprocessed in the following ways:

- The English corpus was tokenized using a modified[1] version of Word Splitter[2].
- All uncapitalized English words were removed.
- Stop words were removed from both corpora (mainly prepositions and auxiliary verbs).
- Any English words of length less than 4 and Arabic words of length less than 3 were removed.

### 4.3.2 Experiment 1: Sentence Aligned Data

The first task I used to test the models was to compare and score the words remaining in each sentence pair after the preprocessing described above. Each algorithm

[1] the way the program handles apostrophes(') had to be modified since they are sometimes used to represent glottal stops in transliterations of Arabic words, e.g. qala'a.

[2] available at http://l2r.cs.uiuc.edu/ cogcomp/tools.php.

| | Arabic | Romanization | English |
|---|---|---|---|
| 1 | مارك | mark | Marks |
| 2 | روسيون | rwsywn | Russian |
| 3 | استراتيجية | istratyjya | strategic |
| 4 | فرنك | frnk | French |

Table 4.4: A sample of the errors made by the word-similarity metrics.

finds the top match for each English word and the top match for each Arabic word. If two words mark each other as their top scorer, then the pair is marked as a transliteration pair. This one-to-one constraint is meant to boost precision, though it will also lower recall. This is because for many of the tasks in which transliteration extraction would be useful (such as building a lexicon), precision is deemed more important. Transliteration pairs are then sorted according to their scores.

The results for the sentence-aligned extraction task are presented in Figure 4.2. Since the number of actual transliterations in the data was unknown, there was no way to compute recall. The measure used here is the precision for each 100 words extracted up to 500.

The baseline, Levenshtein, performs reasonably well. Fuzzy matching is slightly better for the first few hundred words pulled, but quickly drops thereafter. The bootstrapping method is equal to or outperforms the other methods at all levels, including the fuzzy match algorithm which was designed specifically for Arabic. It is particularly impressive that it does not seem to have trouble with digraphs, which I expected would be problematic because of the one-to-one nature of the character operations. Word pairs with two-to-one mappings such as *sh*/ش or *x*/كس tend to score lower than their counterparts composed of only one-to-one mappings, but nevertheless tend to score highly.

A sample of general errors made by all the algorithms is presented in Table 4.4. The most common error was related to inflection (error 1). The words are essentially transliterations of each other, but one or the other of the two words takes a plural or some other inflectional ending that corrupts the phonetic match. The transliterations pairs extracted with these methods are meant to be used as training

| | Metric | Arabic | Romanization | English |
|---|---|---|---|---|
| 1 | Bootstrap | الاخيرين | alakhyryn | Algerian |
| 2 | Bootstrap | وسلم | wslm | Islam |
| 3 | Fuzzy Match | لكل | lkl | Alkella |
| 4 | Fuzzy Match | عمان | 'man | common |
| 5 | ALINE | سكر | skr | sugar |
| 6 | ALINE | اراض | arad | Arab |
| 7 | Levenshtein | واحد | wahd | Wahi |
| 8 | Levenshtein | أصاب | asab | Arab |

Table 4.5: A sample of errors specific to each algorithm.

data for other models, and since these endings could potentially cause problems in generation, I considered them errors. Error 2 represents the common problem of incidental letter similarity. The English *-ian* ending used for nationalities is very similar to the Arabic يون [ijun] and يين [ijin] endings which are used for the same purpose. They are similar phonetically and, since they are functionally similar, will tend to co-occur. Since neither can be said to be derived from the other, however, they cannot be considered transliterations. Error 3 is a case of two words being of common origin but having been modified beyond what would be considered acceptable for a transliteration. Finally, error 4 shows a mapping that would be correct in many transliterations being applied incorrectly. The ك/*c* mapping is supported by many words (including some that would have a ك/*ch* mapping), but leads to an incorrect match in this case.

Algorithm-specific errors indicative of the weaknesses of each metric are presented in Table 4.5. The bootstrapping method encounters problems when erroneous pairs become part of the training data, causing the errors to become reinforced. The only problematic mapping in Error 1 is the خ/*g* mapping, and thus the pair has little trouble getting into the training data. Once the pair is part of training data, the algorithm learns that the mapping is acceptable and uses it to acquire other training pairs that contain the same erroneous mapping. The problem with the fuzzy matching algorithm seems to be that it creates too large a class of equivalent words. Both errors 3 and 4 are given an edit cost of 0. In the case of error 3 this

is due to the letter and vowel normalizations. After letter normalization, the *l*'s are collapsed, leaving *alkela*, then after vowel normalization, it becomes *lkl*, an exact match. Error 4 is due to some of the unusual choices made for letter equivalences. In this case, it is due to the English *c* being considered equivalent to the Arabic ع. ALINE's errors tend to occur when it links two letters, based on phonetic similarity, that are never linked because they each have a more direct equivalent in the other language (errors 5 and 6). For example, in error 5, although the Arabic ك [k] is phonetically similar to the English *g*, they would never be mapped to each other since English has several ways of representing an actual [k] sound. Errors made by Levenshtein distance (errors 7 and 8) are simply due to the fact that it considers all non-identity mappings to be equivalent.

### 4.3.3 Experiment 2: Non-translated Data

The second experiment is meant provide information about how the algorithms would perform on tasks where translations are not available, but the two parts of the bilingual corpus are known to be related. It is meant to loosely mimic the task presented in (Klementiev and Roth, 2006) where the corpora were not necessarily direct translations of each other, but merely comparable (i.e. news articles that describe the same event). The task itself is structured as cross-language named entity recognition (NER). Named entities in the English corpus are tagged using freely available named entity recognition software, and the transliterations of the named entities are searched for in the Arabic translation.

Alternating sentences in the Arabic and English texts were removed, meaning that the texts were no longer direct translations of each other. The English side of the bitext was tagged with Named Entity Tagger[3], which labels named entities as *person*, *location*, *organization* or *miscellaneous*. The words labeled as *person* are extracted. Person names are almost always transliterated, while for the other categories this is far less certain. The list is then hand-checked to ensure that all names are words that are usually transliterated. This left 314 names, though some may not actually appear in the Arabic text due to the sentence removals. The restrictions on

[3]available at http://l2r.cs.uiuc.edu/ cogcomp/tools.php.

| Method | Precision |
|---|---|
| Levenshtein | 42.4 |
| Fuzzy Match | 49.0 |
| ALINE | 55.3 |
| Bootstrapping | 39.5 |

Table 4.6: Precision of the various algorithms on the NER detection task.

| Number of Words | Precision |
|---|---|
| 500 | 0.996 |
| 1000 | 0.996 |
| 1500 | 0.993 |
| 2000 | 0.974 |
| 2500 | 0.962 |
| 3000 | 0.948 |
| Average | 0.978 |

Table 4.7: Results of bootstrapping data extraction from the remainder of the news corpus.

word length and stop words are the same as before, but in this task each of the English person names is compared to all valid words in the Arabic side of the corpus, and the top scorer for each English word is returned.

The results for the NER task are presented in Table 4.6. Obviously, the bootstrapping algorithm performs much more poorly here. It would appear that the algorithm requires some reasonable proportion of the candidates to be potential training examples. Otherwise, it has too many opportunities to corrupt its training data. It is interesting here, however, that ALINE greatly outperforms the fuzzy matching algorithm. It may be that fuzzy matching shares similar requirements to the bootstrapping algorithm due to the large number of word equivalences. Many of the errors it makes are given an edit cost of 0.

## 4.4 Extracting the Training Data

For the generation task to be presented in Chapter 5, I needed data to train the models, and decided that around 3000 words would be sufficient. Since the bootstrap-

ping method performed best on the sentence-aligned data, I ran it on the remainder of the parallel news bitext to extract the training data. The results of this extraction are presented in Table 4.7.

## 4.5 Conclusion

In this chapter, I discussed several metrics of word similarity and evaluated their performance on the task of transliteration extraction. I presented a bootstrapping approach to training a memoriless transducer that learns these values automatically from a bitext. This differs from the other metrics presented which all have the values associated with mapping operations determined a priori. This bootstrapping method is completely language-independent and was shown to outperform the other metrics on a sentence-aligned bitext.

The bootstrapping approach was used to extract the transliteration pairs that will be used to train the generation models to be presented in the following chapter.

# Chapter 5

# Substring-Based Transliteration

Of the two tasks presented in this work, the actual transliteration of an input word is clearly the more difficult. While in the extraction task, one is given a second word to guide the search for a solution, in generating a transliteration, this information is not available. Thus, the ambiguity issues discussed in Chapter 1 become even more pronounced. One must find some way of resolving these ambiguities without the information provided by a second word.

A fundamental question I will be addressing here is how to break the transliteration task down into its basic units. Traditionally, the focus has been to learn to transliterate in units that would be intuitive to a human, be they in terms of phonemes or individual letters. For example, learning mappings such as *sh*/ش or ʃ/ش was considered more important than learning longer unintuitive mappings such as *dal*/دال. This approach is analogous to traditional word-based approaches to **statistical machine translation (SMT)** (Brown *et al.*, 1993), adapted to the setting of transliteration. In recent years, a new approach to SMT, **phrase-based translation** (Koehn *et al.*, 2003), has been found to offer significant improvements over word-based translation models. The phrase-based approach is designed to over-

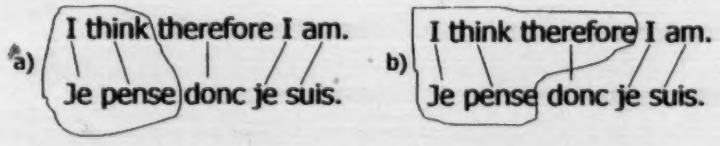


Figure 5.1: Examples of valid (a) and invalid (b) phrase pairs.

come the restrictions on many-to-many mappings in word-based translation models. This approach is based on learning correspondences between phrases, rather than words. Phrases are generated on the basis of a word-to-word alignment, with the constraint that no words within the phrase pair are linked to words outside the phrase pair (Figure 5.1).

I will be looking to apply the phrase-based methodology of machine translation to the domain of transliteration. Section 5.1 presents the noisy channel model, a statistical approach to machine transliteration. Section 5.2 presents my implementation of the letter-based model for Arabic-English transliteration outlined in (Al-Onaizan and Knight, 2002), and also presents a novel many-to-many forward-backward algorithm I designed to train the model. Section 5.3 introduces the **monotone search algorithm** (Zens and Ney, 2004), a linear-time decoding algorithm for phrase-based translation. Section 5.4 presents the substring-based transliteration approach, my adaptation of phrase-based translation methods to machine transliteration. Two models for substring-based transliteration are proposed. The Viterbi substring decoder is a direct adaptation of the monotone search algorithm to transliteration, while the substring-based transducer encodes the substring mapping probabilities into a more flexible finite-state transducer. Section 5.5 presents the experiments performed to test the substring-based models. Comparisons are made to other transliteration methods as well as to a machine translation system (the Google Arabic-English translation Beta).

## 5.1 The Noisy Channel Model

In a statistical approach to machine transliteration, given a foreign word $F$, one is interested in finding the English word $\hat{E}$ that maximizes $P(E|F)$. Modeling $P(E|F)$ is difficult, however, so the task must be broken down into parts that are simpler to model. Using Bayes' rule, and keeping in mind that $F$ is constant, one can formulate the task as follows:

$$\begin{aligned}\hat{E} &= \arg\max_{E} \frac{P(F|E)P(E)}{P(F)} \\ &= \arg\max_{E} P(F|E)P(E)\end{aligned}$$

This is known as the **noisy channel approach** to machine transliteration, which is based on similar models of machine translation (Brown *et al.*, 1990). The noisy channel approach models transliteration generatively in two steps. The English word $E$ is produced with probability $P(E)$, and then is transformed (across the "noisy channel" as it were) into the foreign word $F$ with probability $P(F|E)$. Given a foreign word $F$, the task now is to reverse this process to recover the original English word. The advantage of breaking the task down in this way is that the two probabilities, $P(E)$ and $P(F|E)$, are much easier to model directly than the original $P(E|F)$.

A **language model** is the part of a noisy channel model that provides an estimate of the probability $P(E)$, and it is meant to provide a means for rating strings in terms of how likely they are to be English words. In a comparison to the extraction task, the language model can generally be viewed as attempting to provide information lost by the absence of a second word. The **transliteration model** provides an estimate of the probability $P(F|E)$, and it is meant to rate strings in terms of how likely they are to be the origin of a transliteration $F$. The transliteration model is conceptually similar to the word similarity models discussed in the previous chapter. The probabilities assigned by the transliteration and language models counterbalance each other. For example, simply concatenating the most common mapping for each letter in the Arabic string مايكل, produces the string *maykl*, which is barely pronounceable. To generate the correct *Michael*, a model would need to know the relatively rare letter relationships *ch*/ك and *ae*/ي, and to balance their unlikelihood against an assessment that the correct transliteration is a more probable English name. In terms of the noisy channel model, one hopes a higher $P(E)$ for *Michael* would outweigh its lower $P(F|E)$ enough for it to be selected over *maykl*, despite the latter's higher $P(F|E)$.

The search for the optimal English transliteration $\hat{E}$ for a given foreign name $F$, denoted by the arg max in the above equation, is referred to as decoding. Many decoding algorithms make assumptions about the search space they are exploring, and these assumptions can have a great impact on the optimality of the search, depending on how the search space is defined. A common assumption in many

algorithms is the dynamic programming invariant assumption which states that if the optimal path through a graph happens to go through state $q$, then this optimal path must include the best path up to and including $q$. Thus, once the optimal path to $q$ is known, all other paths to $q$ can be eliminated from the search. The validity of this assumption for a given model depends on how one defines a state $q$. In a dynamic programming approach to transliteration, for example, a state would generally be based on positions in the word being transliterated and its transliteration. If one were transliterating the name مايكل into *Michael*, the path could be defined as going through states <م,*m*>, <اي,*i*>, <ك,*ch*>, and so on. This can be problematic, however. Suppose a dynamic programming model were given the Arabic string كريم, and there are two valid English names in the language model, *Karim* (the correct transliteration of the input) and *Kristine* (the Arabic transliteration of which would be كرستين). The optimal path up to the second letter might go through <ك,*k*>, <ر,*r*>. At this point, it is transliterating into the name *Kristine*, but as soon as it hits the third letter (ي), it is clear that this is the incorrect choice. To recover from the error, the search would have to backtrack to the beginning and return to state <ر,*r*> from a different path, but this is an impossibility since all other paths to that state have been eliminated from the search. Thus it is necessary to define the search space in such a way that the assumptions being made do not hinder the search for an optimal solution.

## 5.2 Letter-based Transliteration

Stalls and Knight (1998) model transliteration from Arabic to English as a sequence of three finite state transducers. The language model P(E) is implemented as a weighted finite state acceptor with probabilities based on word unigram counts from a list of English names. The transliteration model $P(A|E)$ is split into two parts: $P(E'|E)$, the probability of phonetic sequence $E'$ being the pronunciation of English name $E$, and $P(A|E')$, the probability of Arabic word $A$ being a transliteration of $E'$. The $P(E'|E)$ probabilities are learned from the CMU pronunciation dictionary, while the $P(A|E')$ probabilities are learned using the EM algorithm.

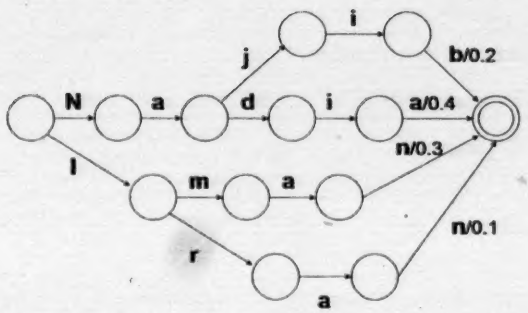


Figure 5.2: A word unigram prefix tree for the names *Najib*, *Nadia*, *Iman* and *Iran*.

Mapping the English word to its phonetic representation seems intuitive at first, since names are generally transliterated based on how they sound, not how they look, but there are problems with this approach. The letter-phoneme conversion itself is not a trivial task. Many transliterated words are proper names, whose pronunciation rules may vary depending on the language of origin (Li *et al.*, 2004). For example, *ch* is generally pronounced as either [ʧ] or [k] in English names, but as [ʃ] in French names. The use of word unigram probabilities as a language model is also problematic because it prevents the model from generating any names that were not seen in the training data.

Al-Onaizan and Knight (2002) address these issues by proposing several modifications to the original model. The transliteration model is implemented as a single transducer that maps directly from English letters to Arabic letters, without the intermediate phonetic conversion. The language model is augmented by adding a letter trigram model to the original word unigram model, allowing for unseen words to be generated.

The word unigram transducer is implemented as a prefix tree (Figure 5.2) just as was proposed in (Knight and Graehl, 1997). However, the details of its augmentation with the letter trigram model in (Al-Onaizan and Knight, 2002) are not made clear, so in my implementation I take its union with the word unigram transducer to create the final language FSA. This model can be seen as taking the max of the word unigram and the letter trigram probabilities to model a given word.

The transliteration transducer presented in (Knight and Graehl, 1997) is a single-

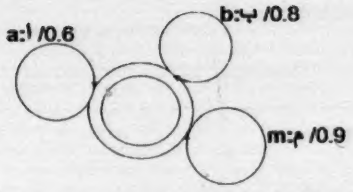


Figure 5.3: A memoriless transliteration transducer.

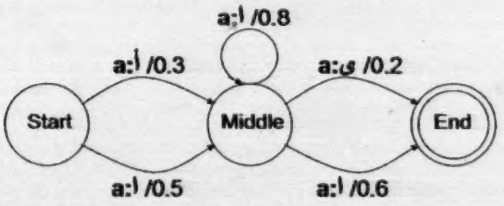


Figure 5.4: A transliteration transducer with separate mapping probabilities for the beginning, middle and end of a word.

state transducer similar to the one presented in Figure 5.3 This structure assumes complete independence of the mapping operations, as it has no means of retaining information about the path taken through it. Stalls and Knight (1998) noted that this independence assumption did not hold, however, as many mappings can be dependent on the positions of letters in the words. For example, the deletion of the *e* in the transliteration of *Joseph* to جوسف would never occur at the beginning of a word. They proposed to have different symbols to represent English phonemes that occur at the beginning, middle or end of a word. This allowed the EM algorithm to learn different mapping probabilities for each part of a word. Al-Onaizan and Knight (2002) found, however, that although the probabilities are learned separately for initial, medial and final phonemes, at generation time, when the input word is Arabic, nothing prevents the model from producing phonemes out of position (e.g. generating an initial phoneme in the middle of a word). They thus proposed to encode the positional dependence into the model by splitting it into three states, as shown in Figure 5.4. The first mapping operation performed in the transliteration takes an arc from the *start* state to the *middle* state, which then loops

back to itself for all medial operations, and final operations occur on arcs to the *end* state.

### 5.2.1 A Many-to-Many Extension to the Forward-Backward Algorithm

The models presented in (Knight and Graehl, 1997) and (Stalls and Knight, 1998) are trained with the following EM algorithm:

1. For each pair of words in the training set, assign uniform probability to each possible alignment of the symbols.
2. For each English phoneme, compute the probability of each symbol it maps to by weighting the mapping counts by the probability of the alignments in which they appear, and then normalizing.
3. For each pair of words, compute the probability of each alignment as the product of the probabilities of the mappings it contains, and normalize the alignment probabilities for each pair of words.
4. Repeat steps 2-3 until convergence.

Al-Onaizan and Knight (2002) use the same algorithm with two modifications. The first is that mappings are learned between English and Arabic letters directly. This is trivial to implement since the algorithm can learn mappings between any two symbol sets. The second is that while the original algorithm learned one-to-many mappings, the modified version can learn many-to-many mappings. To enable it to handle English digraphs and trigraphs, and the occasional diphone, the model learns mappings between 1-3 English letters and 0-2 Arabic letters. This poses a much greater problem in implementation, and it is not made clear how these many-to-many alignments were enumerated for steps 1 and 3 in the above algorithm.

Knight (1999) shows that naive EM algorithms, such as the one presented above, can be implemented as more efficient forward-backward algorithms that perform calculations for several alignments simultaneously. I take this approach to solve the many-to-many mapping problem for transliteration, by adapting the algorithm used

**Algorithm** Expectation-many2many $(s, t)$
**for** $i = 0...I$
  **for** $j = 0...J$
    **if** $(i > 0)$
      **for** $x = 1...3$ **st** $i - x \geq 0$
        $C(s_{i-x+1}^{i}, \epsilon) += \frac{F(i-x,j)P(\epsilon|s_{i-x+1}^{i})B(i,j)}{F(I,J)}$
    **if** $(i > 0 \wedge j > 0)$
      **for** $x = 1...3$ **st** $i - x \geq 0$
        **for** $y = 1...2$ **st** $j - y \geq 0$
          $C(s_{i-x+1}^{i}, t_{j-y+1}^{j}) += \frac{F(i-x,j-y)P(t_{j-y+1}^{j}|s_{i-x+1}^{i})B(i,j)}{F(I,J)}$

Figure 5.5: Pseudocode for a many-to-many expectation algorithm.

**Algorithm** EM-many2many

**for all** mapping operations $(a, b)$
  $C(a, b) := 0$
**for each** sequence pair $(s, t)$
  *Expectation-many2many*$(s, t)$
*Maximization-Step*$(C)$

Figure 5.6: Pseudocode for a many-to-many expectation maximization algorithm.

to train a one-to-one stochastic transducer in (Ristad and Yianilos, 1998) (Chapter 2) to a many-to-many framework.

Partial counts for each mapping operation are collected in the $C$ table. For each English-Arabic training pair $(s, t)$ the *EM-many2many* function (Figure 5.6) calls the *Expectation-many2many* function (Figure 5.5) to collect partial counts. $I$ and $J$ are the lengths of English name $s$ and Arabic name $t$, respectively. The *Maximization-step* function simply normalizes the partial counts to create a conditional probability distribution.

The *Forward-many2many* function (Equation 5.1) fills in the table $F$, with each entry $F(i, j)$ being the sum of the probabilities all possible alignments of the prefix pair $(s_1^i, t_1^j)$. Analogously, the *Backward-many2many* function (Equation 5.2) fills in table $B$, with each entry $B(i, j)$ being the sum of all paths through the transducer that generate the suffix pair $(s_{i+1}^I, t_{j+1}^J)$.

$$
\begin{aligned}
F(0,0) &= 1 \\
F(i,j) &= \sum_{\substack{1\le x\le 3,\\ 1\le y\le 2}} P(\epsilon|s_{i-x+1}^{i})F(i-x,j) + P(t_{j-y+1}^{j}|s_{i-x+1}^{i})F(i-x,j-y)
\end{aligned}
\tag{5.1}
$$

$$
\begin{aligned}
B(I,J) &= 1 \\
B(i,j) &= \sum_{\substack{1\le x\le 3,\\ 1\le y\le 2}} P(\epsilon|s_{i+1}^{i+x})B(i+x,j) + P(t_{j+1}^{j+y}|s_{i+1}^{i+x})B(i+x,j+y)
\end{aligned}
\tag{5.2}
$$

*Expectation-many2many* uses the probabilities in the $F$ and $B$ tables to calculate partial counts for every possible mapping in the pair of names. The partial count collected at positions $i$ and $j$ in the sequence pair is sum of all paths that generate the sequence pair and go through $(i, j)$, divided by the sum of all paths that generate the entire sequence pair ($F(I, J)$).

The modifications to the one-to-one forward-backward algorithm presented in Chapter 2 should be readily apparent. The many-to-many extension lies in the *for* loops that iterate over $x$ and $y$. These allow the algorithm to consider several combinations of letters from the source and target words when calculating probabilities or collecting counts. Some parameters are specified to match those proposed in in Al-Onaizan and Knight (2002). A general version of the algorithm is presented in Appendix A.

## 5.3 The Monotone Search Algorithm

Zens and Ney (2004) propose a linear-time decoding algorithm for phrase-based machine translation. The algorithm requires that the translation of phrases be sequential, disallowing any phrase reordering in the translation.

Starting from a word-based alignment for each pair of sentences, the training for the algorithm accepts all contiguous bilingual phrase pairs (up to a predetermined maximum length) whose words are only aligned with each other (Koehn *et al.*, 2003). The probabilities $P(\tilde{f}|\tilde{e})$ for each foreign phrase $\tilde{f}$ and English phrase $\tilde{e}$ are calculated on the basis of counts gleaned from a bitext. Since the counting process is much simpler than trying to learn the phrases with EM, the maximum phrase length can be made arbitrarily long with minimal jumps in complexity. This allows the

model to actually encode contextual information into the translation model instead of leaving it completely to the language model. There are no null ($\epsilon$) phrases so the model does not handle insertions or deletions explicitly. They can be handled implicitly, however, by including inserted or deleted words as members of a larger phrase.

Decoding in the monotone search algorithm is performed with a Viterbi dynamic programming approach. For a foreign sentence of length $J$ and a phrase length maximum of $M$, a table is filled with a row $j$ for each position in the input foreign sentence, representing a translation sequence ending at that foreign word, and each column $e$ represents possible final English words for that translation sequence. Each entry in the table $Q$ is filled according to the following recursion:

$$\begin{aligned} Q(0,\$) &= 1 \\ Q(j,e) &= \max_{e',\tilde{e},\tilde{a}} P(\tilde{a}|\tilde{e})P(\tilde{e}|e')Q(j',e') \\ Q(J+1,\$) &= \max_{e'} Q(J,e')P(\$|e') \end{aligned}$$

where $\tilde{f}$ is a foreign phrase ending at $j$ and consisting of up to $M$ words. The '\$' symbol is the sentence boundary marker.

In the above recursion, the language model is represented as $P(\tilde{e}|e')$, the probability of the English phrase given the previous English word. Because of data sparseness issues in the context of word phrases, the actual implementation approximates this probability using word n-grams.

## 5.4 Substring-based Transliteration

I propose to adapt phrase-based models of statistical machine translation to the task of machine transliteration by transliterating on the basis of substrings rather than individual letters. There are several apparent advantages to this approach over the letter-based approach described in Section 5.2.

- The longer substrings in the mapping operations allow for some modeling of contextual dependencies to be encoded into the transliteration model. Fur-

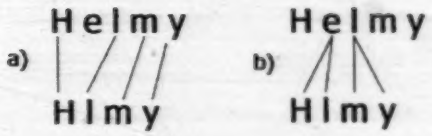


Figure 5.7: High-probability (a) and low-probability (b) alignments of *Helmy* and حلمي. The Arabic is Romanized for clarity.

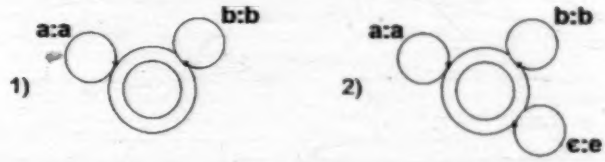


Figure 5.8: Transducers without (1) and with (2) nulls allowed in the input word.

thermore, since the longer English substrings are known to have been seen in the data, they can be assumed to be well-formed.

- The letter-based EM training considers all possible alignments of a given training pair, even those that are extremely unlikely (Figure 5.7). This means that the model will end up encoding many low-probability mappings. This has implications in terms of the size of the transducers created as well as on the quality of transliterations, depending on how these low-probability mappings interact with the language model.

- In terms of complexity, the absence of explicit nulls on the input side has major implications on the complexity of the transducers created. For example, if transducer *1* in Figure 5.8 were given the string *aab*, the only possible output would be *aab*. On the other hand, if the same string were input into transducer *2*, the output could be *aab*, *eaab*, *aeab*, *eeaeabe*, and so on. The number of potential output strings is, in fact, infinite. In the case of transliteration, where this transducer must be composed with a language transducer, the problem becomes especially pronounced. The composition must be done in such a way that there is a valid path through the full transducer for any potential string that could be output by the transliteration transducer and has

**Algorithm** VS-Decode $(A, max)$

$Q(0, \$) := 1$
**for** $i := 1$ **to** length$(A)$ **do**
  **for** $j := 1$ **to** $max$ **do**
    $seg := A_{i-j..i}$
    **for all** $e'$ **st** $Q(i-j-1, e') > 0$ **do**
      **for all** $\tilde{e}$ **st** $P(seg|\tilde{e}) > 0$ **do**
        $\mathbf{e} := \tilde{e}_{length(\tilde{e})}$
        **if** $P(seg|\tilde{e})P(\tilde{e}|e')Q(i-j-1, e') > Q(i, e)$
          $[Q(i, e) := P(seg|\tilde{e})P(\tilde{e}|e')Q(i-j-1, e')]$

**for all** $e'$ **st** $Q(length(A), e') > 0$ **do**
  **if** $P(\$|e')Q(length(A), e') > Q(length(A)+1, \$)$
    $[Q(length(A)+1, \$) := P(\$|e')Q(length(A), e')]$

Figure 5.9: The Viterbi substring decoding algorithm.

a positive probability in the language transducer. This leads to prohibitively large transducers. The substring-based models handle nulls implicitly (e.g. the mapping *ke*:ك implicitly represents *e*:ε after a *k*), so the model itself is not required to deal with them.

With these issues in mind, I propose to model substring-based transliteration in two ways. The first, the Viterbi substring decoder, is a direct adaptation of the monotone search algorithm, described in Section 5.3, to the domain of transliteration. The second, the substring-based transducer, encodes the substring-based transliteration model as a transducer and composes it with the word unigram/letter trigram language model described in Section 5.2.

### 5.4.1 Viterbi Substring Decoder

The Viterbi substring decoder is a straightforward adaptation of the monotone search algorithm to the domain of transliteration, in which letters and word substrings are substituted for the words and phrases of the original model. Pseudocode for the algorithm is presented in Figure 5.9. There are, in fact, strong indications that the monotone search algorithm is better suited to transliteration than it is to translation.

Figure 5.10: A one-to-one alignment of *Mourad* and مراد. For clarity the Arabic name is written left to right.

Unlike machine translation, where the constraint on reordering required by monotone search is frequently violated, transliteration is an inherently sequential process. Also, the sparsity issue in training the language model is much less pronounced, allowing $P(\tilde{e}|e')$ to be modeled directly.

In order to train the model, I extract the one-to-one Viterbi alignment of a training pair from a stochastic transducer as discussed in Chapter 2. Substrings are then generated by iteratively appending adjacent links or unlinked letters to the one-to-one links of the alignment. For example, assuming a maximum substring length of 2, the <*r*, ر> link in the alignment presented in Figure 5.10 would participate in the following substring pairs: <*r*, ر>, <*ur*, ر>, and <*ra*, را>.

### 5.4.2 Substring-based Transducer

One major advantage the letter-based transducer presented in Section 5.2 has over the Viterbi substring decoder is its word unigram language model. This type of language model cannot be added to the Viterbi substring decoder, because of the way states are defined in its search ($Q(i, e)$ in Figure 5.9), and the dynamic programming invariant assumption made by the Viterbi algorithm. On the other hand, the Viterbi substring decoder is able to encode contextual information in the transliteration model because of its ability to consider larger many-to-many mappings. In a novel approach presented here, I propose a substring-based transducer that draws on both advantages. The substring transliteration model learned for the Viterbi substring decoder is encoded as a transducer, whose state-space is defined in a more flexible way than that of the Viterbi substring decoder. FSTs (e.g. Figures 5.2, 5.3 and 5.4), are clearly not required to define states based on letters in the input or output. Thus, the backtracking problem in the *Karim/Kristine* example mentioned

| | Letter Transducer | Viterbi Substring | Substring Transducer |
|---|---|---|---|
| Model Type | Transducer | Dynamic Prog. | Transducer |
| Trans. Model | Letter | Substring | Substring |
| Lang. Model | Word/Letter | Substring/Letter | Word/Letter |
| Null Symbols | Yes | No | No |
| Alignments | All | Most Probable | Most Probable |

Table 5.1: Comparison of statistical transliteration models.

earlier can be avoided, even if the same search assumptions are made.

Compared to the letter-based transducer, the more structured approach to learning mapping operations should lead to higher quality transliterations, while the absence of explicit nulls should have a major impact on efficiency. The substring-based transducer should also have an advantage over the Viterbi substring decoder because of its stronger word unigram language model. An overview of the three statistical transliteration models discussed in this chapter is presented in Figure 5.1.

## 5.5 Experiments

In this section, I describe the evaluation of my models on the task of Arabic-to-English transliteration.

### 5.5.1 Data

For my experiments, I required bilingual name pairs for testing and development data, as well as for the training of the transliteration models. To train the language models, I simply needed a list of English names. Bilingual data was extracted according to methods described in Chapter 4. After extraction, the bilingual training data was reviewed by hand to remove any non-transliterations. The English name list for the language model training was extracted from the English-Arabic Treebank v1.0 (approx. 52k words), available from the Linguistic Data Consortium. Despite the name, this corpus contains only English data. The language model training set consisted of all words labeled as proper names in this corpus along with all the English names in the transliteration training set. Any names in any of the

data sets that consisted of multiple words (e.g. first name/last name pairs) were split and considered individually.

Training data for the transliteration model consisted of 2844 English-Arabic pairs. The language model was trained on a separate set of 10991 (4494 unique) English names. The final test set of 300 English-Arabic transliteration pairs contained no overlap with the set that was used to induce the transliteration models. With respect to the language model, the English side of the test set contained 146 *seen* and 154 *unseen* names. I report results separately for each category. The unseen names were mostly of Arabic origin, while seen names tended to be non-Arabic.

### 5.5.2 Evaluation Methodology

The Arabic names in the test set served as input to the models, while the English names in the set were considered gold standard transliterations for the purpose of evaluation. The transliterated output was compared to the gold standard according to two metrics. The first metric was the exact match accuracy with respect to the gold standard. This is an extremely strict measure of correctness, particularly in the case of forward transliteration. There are often several correct English spelling variations of Arabic names, but only one will match the gold standard. The second metric, average Levenshtein distance, is a softer standard of correctness and gives a sense of how far off the mark the incorrect transliterations actually are. It is computed as the minimum number of insertions, deletions and substitutions between a proposed transliteration and the gold standard, averaged over all pairs in the test set. It should be noted that Levenshtein distance as metric may still not be fine-grained enough in some cases. For example, if the gold standard transliteration were *Mahmud*, its Levenshtein distance from *Mahmood* and *Mahfuz* is exactly the same. The problem is that vowel substitutions, which are generally insignificant in terms of correctness, are considered equivalent to consonant substitutions, which usually are significant. This problem could be alleviated by manually assigning lower costs to certain substitutions in the distance calculation, but this would involve making fairly arbitrary decisions about the costs. In general, I wished to avoid these types of decisions in evaluating the transliteration models.

I performed a second test with the statistical transliteration models on words that appeared in both the transliteration and language model training data. This test was not indicative of the overall strength of the models but was meant to give a sense of how much each model depends on its language model versus its transliteration model.

Finally, I performed a third test to give a sense of how the substring-based models would affect a machine translation system. The test words from the first experiment were input into the Google Arabic to English translator Beta[1]. The words were input in two ways. The first was to input bare words, just as was done for the transliterations models, and the second was to input the names as part of a template sentence to provide the translations system with some context. It was found, however, that the performance on bare words was slightly better, since the translation system occasionally combined context words with the word to be transliterated when making the translation. Thus the results presented will be for bare word input.

### 5.5.3 Setup

Five approaches were evaluated on the Arabic-English transliteration task.

- **Baseline:** As a baseline for my experiments, I used a simple deterministic mapping algorithm which maps Arabic letters to the most likely corresponding letter or sequence of letters in English.

- **Letter-based Transducer:** The letter-based transducer is the model presented in Section 5.2. The transducer was implemented in Carmel[2].

- **Viterbi Substring Decoder:** The Viterbi substring decoder is the model presented in Section 5.4.1. I tested maximum substring lengths between 3 and 10 and found that a maximum length of 6 was optimal on the development data.

[1] http://www.google.ca/language_tools?hl=en

[2] Carmel is a finite-state transducer package written by Jonathan Graehl. It is available at http://www.isi.edu/licensed-sw/carmel/.

| Method | Seen | Unseen |
|---|---|---|
| Baseline | 0.7 | 3.9 |
| Letter transducer | 57.5 | 2.6 |
| Viterbi substring | 27.4 | **9.7** |
| Substring transducer | **75.3** | 3.9 |
| Human | 44.5 | 29.2 |

Table 5.2: Exact match accuracy percentage on the test set for various methods.

| Method | Seen | Unseen |
|---|---|---|
| Baseline | 2.77 | 2.42 |
| Letter transducer | 1.30 | 2.49 |
| Viterbi substring | 1.52 | **1.79** |
| Substring transducer | **0.66** | 2.03 |
| Human | 1.30 | 1.35 |

Table 5.3: Average Levenshtein distance on the test set for various methods.

- **Substring-based Transducer:** I implemented the substring-based transducer by encoding the substring-based transliteration model into a transducer as outlined in Section 5.4.2, and composing it with the language transducer used in the letter-based transducer. The substring-based transducer was also implemented in Carmel. I found that this model worked best with a maximum substring length of 4.

- **Human:** For the purpose of comparison, I allowed an independent human subject (fluent in Arabic, but a native speaker of English) to perform the same task. The subject was asked to transliterate the Arabic words in the test set without any additional context. No additional resources or collaboration were allowed.

### 5.5.4 Results on the Test Set

Table 5.2 presents the word accuracy performance of each transliterator in terms of exact matches. Table 5.3 shows the average Levenshtein distance results. The results of the human subject show that this task is not trivial, even for humans,

| | Arabic | LBT | SBT | Correct |
|---|---|---|---|---|
| 1 | عثمان | Uthman | Uthman | Othman |
| 2 | اشرف | Asharf | Asharf | Ashraf |
| 3 | رفعت | Rafeet | Arafat | Refaat |
| 4 | اسامة | Istamaday | Asuma | Usama |
| 5 | ايمان | Erdman | Aliman | Iman |
| 6 | ووتش | Wortch | Watch | Watch |
| 7 | ميلز | Mellis | Mills | Mills |
| 8 | فيراري | February | Firari | Ferrari |

Table 5.4: A sample of the errors made by the letter-based (LBT) and substring-based (SBT) transducers.

and thus give a reasonable expectation of performance given the standards used for testing here. Obviously, the seen/unseen distinction is only relevant for models that employ a language model. The human subject and the baseline do not get to see any data, so the test data is all the same with respect to them, but their performance on each subset give some standards for comparison and indicate that obtaining exact matches is more difficult on words in the unseen category.

Overall, the substring-based transducer that I propose clearly outperforms the letter-based transducer. Its accuracy is better in both categories, but its advantage is particularly pronounced on words it has seen in the training data for the language model (the task for which the letter-based transducer was originally designed). Since both transducers use exactly the same language model, the fact that the substring-based transducer outperforms the letter-based transducer indicates that it learns a stronger transliteration model.

The word unigram language model used in the letter- and substring-based transducers greatly boosts performance for the seen words. The differences in performance between the Viterbi substring decoder and the substring-based transducer suggest that the substring-based language model used by the former is fairly weak when it comes to recreating words it has seen before, though it is stronger than the letter trigram model for unseen words.

A sample of the errors made by the letter- and substring-based transducers is

| Method | Time | Size (states/arcs) |
|---|---|---|
| Letter transducer | 5h52min | 84490/538834 |
| Substring transducer | 11 sec | 735/2094 |

Table 5.5: Running times and transducer sizes for a typical input word.

presented in Table 5.4. In general, when both models err, the substring-based transducer tends toward more phonetically reasonable choices. The most common type of error is simply correct alternate English spellings of an Arabic name (error 1), since they do not match the gold standard. Error 2 is an example of a learned mapping being misplaced (the deleted *a*), though the letter-based transducer is able to avoid this at the beginning or end of a word because of its three-state transliteration transducer (error 3). Errors 6 and 7 are names that actually appear in the word unigram model but were missed by the letter-based transducer, while error 8 is an example of the letter-based transducer incorrectly choosing a name from the word unigram model. As discussed in Section 5.4, this is likely due to mappings learned from low-probability alignments.

### 5.5.5 Computational Considerations

Another point of comparison between the statistical transliteration models is complexity. The running times and transducer sizes for both transducer-based approaches are presented in Table 5.5. The running time for the Viterbi substring decoder was 3 sec. Tests were performed on an AMD Athlon 64 3500+ machine with 2GB of memory running Red Hat Enterprise Linux release 4. Running times are for the 300 word test set. The sizes presented are for composition with a sample name from the test set, حلمي (*Helmy*). The letter-based transducer encodes 56144 mappings while the substring-based transducer encodes 13948, but the size difference between the transducers created is clearly not proportional to this. As discussed in Section 5.4, the reason for the size explosion factor in the letter-based transducer is the possibility of null characters in the input word.

| Method | Exact match | Average Levenshtein |
|---|---|---|
| Letter transducer | 81.2 | 0.46 |
| Viterbi substring | 83.2 | 0.24 |
| Substring transducer | **94.4** | **0.09** |

Table 5.6: Results for testing on the transliteration training set.

| Method | Seen | Unseen |
|---|---|---|
| Viterbi substring | 27.4 | 9.7 |
| Substring transducer | **75.3** | 3.9 |
| Google | 58.2 | **33.8** |

Table 5.7: Comparison of substring transliterators to the Google translator in terms of exact match.

### 5.5.6 Results on the Training Set

The substring-based approaches encode a great deal of contextual information into the transliteration model. In order to assess how much the performance of each approach depends on its language model versus its transliteration model, I tested the three statistical models on the set of 2844 names seen in both the transliteration and language model training. The results of this experiment are presented in Table 5.6. The Viterbi substring decoder receives the biggest boost, outperforming the letter-based transducer, which indicates that its strength lies mainly in its transliteration modeling as opposed to its language modeling. The substring-based transducer, however, still outperforms it by a large margin, achieving near-perfect results. Most of the remaining errors can be attributed to names with alternate correct spellings in English.

### 5.5.7 Comparison to a Machine Translation System

The results for the comparison with the Google translator are shown in Figures 5.7 and 5.8. It is difficult to draw concrete conlusions since the translator is performing a different task. It should also be noted that the underlying methods and algorithms used by Google, as well as its training data, are unknown. However, simply comparing the numbers, one can see, first of all, that the substring-based transducer

| Method | Seen | Unseen |
|---|---|---|
| Viterbi substring | 1.52 | **1.79** |
| Substring transducer | **0.66** | 2.03 |
| Google | 1.08 | 3.24 |

Table 5.8: Comparison of substring transliterators to the Google translator in terms of average Levenshtein distance.

| | Arabic | Google | SBT | Correct |
|---|---|---|---|---|
| 1 | السيد | Mr. | Alsaid | AlSayed |
| 2 | فتح | Open | Fatah | Fatah |
| 3 | كامل | Full | Kamal | Kamel |
| 4 | فروهار | Forouhar | Fruhar | Fruhar |
| 5 | تلتسكوي | Tltscui | Taltskawy | Teletskoye |
| 6 | كيميروفو | - | Kimirovo | Kimirovo |
| 7 | بافس | London:Macmillan. | Pavis | Puffs |
| 8 | خضوري | The Palestinian Territories, headed | Khadwari | Khadori |

Table 5.9: A sample of the errors made by the Google translator.

outperforms the translator on words seen in the transducer's language training. Another point of interest is that on the unseen words (with respect to the substring-based models), despite the fact that the translator performs better in terms of exact matches, it performs much worse in terms of average Levenshtein distance. This means that although the translator is achieving more exact matches, when it does err, it errs quite badly. This is clearly due to the systems's tendency to translate instead of transliterate, since many Arabic names also have meanings as common nouns.

A sample of the errors made by the Google translator is shown in Table 5.9. Errors 1-3 show the basic error made by the translator of translating a word instead of transliterating it. Errors 4 and 5 suggest that there is some type of transliteration being done by the system, since the outputs are non-words that resemble the input. However, error 6 shows a case where the translator was unable to produce any output, despite the fact that there are no uncommon mappings in the transliteration. Errors 7 and 8 show exceptional cases where the error cannot be explained

by transliteration or translation errors. The output does not resemble the input in any way. These may simply be the result of anomolous mappings learned in the system's training, though it is impossible to be certain.

## 5.6 Conclusion

My substring-based approach to machine transliteration borrows concepts from phrase-based models of translation and applies them to the task of transliteration. In this chapter, I presented two models to implement this approach. The Viterbi substring decoder uses dynamic programming to find the optimal transliteration for a given Arabic word. It is a direct adaptation of the monotone search algorithm for translation to the domain of transliteration. The substring-based transducer uses the same transliteration model as the Viterbi substring decoder, but encodes it as a transducer. This allows for a more flexible choice in terms of the language models it can use.

The main point of comparison for these new models was the letter-based transducer presented in (Al-Onaizan and Knight, 2002). I proposed a novel many-to-many extension to the forward-backward algorithm presented in (Ristad and Yianilos, 1998), and used this many-to-many algorithm to train the letter-based model.

I showed that the substring-based approach is able to significantly outperform the letter-based transducer, and at the same time be orders of magnitude less complex. When errors were made by both the letter-based and substring-based models, the substring-based model tended towards more phonetically reasonable errors. If the model were being used in a system to produce transliterations that could be checked by a human, this would obviously make it more useful than the letter-based approach. I presented several advantages to this approach over traditional letter-based or phoneme-based models of transliteration. The ability to commit to a simple high-probability alignment makes training less complex, and also prevents it from learning low-probability mappings that lead to errors in the letter-based model. The exclusion of explicit null characters in the model leads directly to significant gains in computational efficiency.

# Chapter 6

# Conclusion

The main contribution of this work was the introduction of a new framework for generating transliterations, the substring-based approach. Substring-based models learn mappings between word substrings, rather than individual letters or phonemes, in an approach analogous to phrase-based models of machine translation. I presented two models of substring-based transliteration. The Viterbi substring decoder is a dynamic programming approach, and the substring-based transducer models transliteration as a series of transducers. I also presented a many-to-many extension to the forward-backward algorithm and used it to train a letter-based transducer for the purpose of comparison. The substring-based models were shown to outperform the letter-based model and at the same time offer substantial savings in computational complexity.

Since my approach is data-driven, I required examples of transliteration, and this led me to explore avenues for extracting these examples automatically from a bitext. I presented a bootstrapping approach to training a memoriless transducer for the task. This method learned the relationships between letters automatically from the bitext, and thus required no a priori language knowledge to perform its evaluations. It was found to outperform traditional models of word similarity on a sentence-aligned bitext and was used to gather training and testing data required for the statistical transliteration models.

Though the generation and extraction tasks differ in many significant ways, I approached both with certain common goals in mind. Besides the obvious goal of strong performance, I also wished to design my models to be as language-

independent as possible so that they might be ported to other language pairs with minimal retooling. The models presented in this work were tested on Arabic-English transliteration but could easily be ported to other language pairs. Since tests on other languages have not yet been run, however, I can only claim that the proposed models could be applied to other languages, but not that they would perform as strongly. The substring-based models should not have much trouble in transliterating between other language pairs, even those in non-alphabetic scripts, since they can handle many-to-many relationships of arbitrary length. On the other hand, the one-to-one nature of the bootstrapped transducer may make its application to language pairs with more complex many-to-many relationships more difficult. Thus, one of the key areas for future work would be to test the proposed models on other language pairs.

One of the more promising avenues for future work in the generation of transliterations would be the use of methods such as **discriminative reranking** (Och and Ney, 2002) to improve transliteration accuracy. Discriminative reranking is a method used in machine translation. Features are extracted from an n-best list of translation candidates proposed by some translation model for a given input sentence. Based on these features, candidates are reranked, and the new top-ranked candidate is selected as the translation. The transducers presented in Chapter 5 are easily able to produce an n-best list of transliteration candidates so the application of discriminative reranking should be straightforward.

I also plan to assess how well the substring-based approach applies to other generative tasks. Some preliminary tests have been made on applying it to letter-to-phoneme conversion, and I would also like explore other areas such as spelling correction.

The question of how to evaluate generated transliterations also remains open. Since the concept of a "correct" transliteration remains ill-defined, it is difficult to devise a completely empirical measure by which to evaluate proposed transliterations. As mentioned in Chapter 5, even Levenshtein distance, as a metric, is not nuanced enough to capture what a human might consider correct. One approach might be to modify the Levenshtein distance metric, perhaps along the lines of the

fuzzy matching algorithm presented in Chapter 4, in order to draw the line between correct and incorrect more clearly. Another possibility would be to simply allow human judges to make the distinction. Since both measures would involve subjective decisions, however, there may be some question as to whether they can be considered empirical.

In detecting transliterations, the main area open to future research would be the use of additional information from the bitext to improve precision. Since one of my main goals was to make the model as language independent as possible, the addition of any language knowledge is an unattractive option. Information such as word frequency or distribution in the bitext could certainly be incorporated into my model to improve performance.

# Bibliography


N. AbdulJaleel and L. S. Larkey. Statistical transliteration for English-Arabic cross language information retrieval. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 139–146, New York, NY, USA, 2003. ACM Press.

Y. Al-Onaizan and K. Knight. Machine transliteration of names in Arabic text. In *Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages*, 2002.

M. Arbabi, S.M. Fischthal, V.C. Cheng, and E. Bart. Algorithmns for Arabic name transliteration. *IBM Journal of Research and Development*, 38(2), 1994.

L. E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of markov processes. *Inequalities*, 3:1–8, 1972.

P. F. Brown, J. Cocke, S. Della Pietra, V. J. Della Pietra, F. Jelinek, J. D. Lafferty, R. L. Mercer, and P. S. Roossin. A statistical approach to machine translation. *Computational Linguistics*, 16(2):79–85, 1990.

P. F. Brown, V. J. Della Pietra, S. A. Della Pietra, and R. L. Mercer. The mathematics of statistical machine translation: parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

N. Collier, A. Kumano, and H. Hirakawa. Acquisition of English-Japanese proper nouns from noisy-parallel newswire articles using Katakana matching. In *Natural Language Pacific Rim Symposium (NLPRS'97), Phuket, Thailand*, pages 309–314, December 2–4 1997.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

A. Ekbal, S. K. Naskar, and S. Bandyopadhyay. A modified joint source-channel model for transliteration. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 191–198, Sydney, Australia, July 2006. Association for Computational Linguistics.

A. Freeman, S. Condon, and C. Ackerman. Cross linguistic name matching in English and Arabic. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 471–478, New York City, USA, June 2006. Association for Computational Linguistics.

A. Klementiev and D. Roth. Named entity transliteration and discovery from multilingual comparable corpora. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 82–88, New York City, USA, June 2006. Association for Computational Linguistics.

K. Knight and J. Graehl. Machine transliteration. In *Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 128–135, Somerset, New Jersey, 1997. Association for Computational Linguistics.

K. Knight. Decoding complexity in word-replacement translation models. *Computational Linguistics*, 25(4):607–615, 1999.

P. Koehn, F. J. Och, and D. Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*, pages 48–54, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

G. Kondrak. A new algorithm for the alignment of phonetic sequences. In *Proceedings of NAACL 2000*, pages 288–295, 2000.

C. Lee and J. S. Chang. Acquisition of English-Chinese transliterated word pairs from parallel-aligned texts using a statistical machine transliteration model. In *Proceedings of the HLT-NAACL 2003 Workshop on Building and using parallel texts*, pages 96–103, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966.

H. Li, M. Zhang, and J. Su. A joint source-channel model for machine transliteration. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 159–166, Barcelona, Spain, July 2004.

I. D. Melamed. Automatic construction of clean broad-coverage translation lexicons. In *Second Conference of the Association for Machine Translation in the Americas*, pages 125–134, 1996.

F. J. Och and H. Ney. Improved statistical alignment models. In *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 440–447, Morristown, NJ, USA, 2000. Association for Computational Linguistics.

F. J. Och and H. Ney. Discriminative training and maximum entropy models for statistical machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 295–302, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

E. S. Ristad and P. N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

B. Stalls and K. Knight. Translating names and technical terms in Arabic text. In *Proceedings of the COLING/ACL Workshop on Computational Approaches to Semitic Languages*, 1998.

K. Tsuji. Automatic extraction of translational Japanese-katakana and English word pairs. *International Journal of Computer Processing of Oriental Languages*, 15(3):261–279, 2002.

A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, 1967.

R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

D. Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Meeting of the Association for Computational Linguistics*, pages 189–196, 1995.

R. Zens and H. Ney. Improvements in phrase-based statistical machine translation. In *Proceedings of the Human Language Technology Conference (HLT-NAACL)*, pages 257–264, Boston, MA, May 2004.

# Appendix A

# The Many-to-Many Forward-Backward Algorithm

The following is a general version of the many-to-many forward-backward algorithm presented in Chapter 5. The algorithm presented in Chapter 5 specified parameters to match the training parameters outlined in (Al-Onaizan and Knight, 2002). The algorithm here differs in the following ways:

- Insertions are allowed.
- The size of substrings involved in mapping operations can be of any size. The $smax$ and $tmax$ variables represent the maximum substring sizes for the source and target words respectively.
- The conditional probability $P(t_j|s_i)$ has been replaced by the function $\delta(s_i, t_j)$, which could represent either a joint or conditional distribution, depending on how the *Maximization-Step* function is defined.

The algorithm can also be compared to the one-to-one forward-backward algorithm presented in Chapter 2. The core of the extension lies in the *for* loops iterating over $x$ and $y$, which allow the algorithm to consider various combinations of substrings when building up the $F$ and $B$ tables.

**Algorithm** EM-many2many $(smax, tmax)$

**for all** mapping operations $(a, b)$
  $C(a, b) := 0$
**for each** string pair $(s, t)$
  *Expectation-many2many*$(s, t, smax, tmax)$
*Maximization-Step*$(C)$

Figure A.1: Pseudocode for a general many-to-many expectation maximization algorithm.

**Algorithm** Expectation-many2many $(s, t, smax, tmax)$
**for** $i = 0...I$
  **for** $j = 0....J$
    **if** $(i > 0)$
      **for** $x = 1...smax$ **st** $i - x \geq 0$
        $C(s^i_{i-x+1}, \epsilon) += \frac{F(i-x,j)\delta(s^i_{i-x+1},\epsilon)B(i,j)}{F(I,J)}$
    **if** $(j > 0)$
      **for** $y = 1...tmax$ **st** $i - x \geq 0$
        $C(\epsilon, t^j_{j-y+1}) += \frac{F(i,j-y)\delta(\epsilon,t^j_{j-y+1})B(i,j)}{F(I,J)}$
    **if** $(i > 0 \wedge j > 0)$
      **for** $x = 1...smax$ **st** $i - x \geq 0$
        **for** $y = 1...tmax$ **st** $j - y \geq 0$
          $C(s^i_{i-x+1}, t^j_{j-y+1}) += \frac{F(i-x,j-y)\delta(s^i_{i-x+1},t^j_{j-y+1})B(i,j)}{F(I,J)}$

Figure A.2: Pseudocode for a general many-to-many expectation algorithm.

$$F(0,0) = 1$$

$$F(i,j) = \sum_{\substack{1\le x\le smax,\\ 1\le y\le tmax}} \begin{cases} & \delta(s^i_{i-x+1},\epsilon)F(i-x,j) \\ + & \delta(\epsilon,t^j_{j-y+1})F(i,j-y) \\ + & \delta(s^i_{i-x+1},t^j_{j-y+1})F(i-x,j-y) \end{cases} \tag{A.1}$$

$$B(I,J) = 1$$

$$B(i,j) = \sum_{\substack{1\le x\le smax,\\ 1\le y\le tmax}} \begin{cases} & \delta(s^{i+x}_{i+1},\epsilon)B(i+x,j) \\ + & \delta(\epsilon,t^{j+y}_{j+1})B(i,j+y) \\ + & \delta(s^{i+x}_{i+1},t^{j+y}_{j+1})B(i+x,j+y) \end{cases} \tag{A.2}$$

# Glossary

**abjad** Alphabet in which most vowels are left unwritten.

**alignment** A linking of characters between two strings.

**ALINE** An algorithm for calculating the similarity and producing an alignment between two phonetic strings.

**alphabetic script** Script which uses symbols to represent phonemes in a given language.

**back transliteration** The recovery of a previously transliterated name to its native writing script.

**bootstrapping** A weakly supervised approach to training a model in which a small set of labeled seed examples is used to learn initial parameters. The initial parameters are then used to extend the training set by labeling unlabeled examples. This process is repeated and the training set is extended iteratively until some stopping condition has been satisfied.

**composition** An operation on a set of finite-state transducers in which the output from one transducer is used as input for the next.

**data-driven** Class of algorithms which learn to function based on examples of the task to be performed.

**decoding** The extraction of information from a model based the model parameters and the input.

**deletion** Edit operation in which a character or group of characters in the source string are not mapped to anything in the target string.

**Dijkstra's algorithm** A search algorithm based on maintaining the shortest path to any state visited so far and always taking the shortest arc outward from the periphery of the search.

**discriminative reranking** An approach used in machine translation in which a primary translation model produces a list of candidates for the translation of an input sentence, and the candidates are reranked based on features that are suggested to mark a "good" translation.

**dynamic programming** Class of table based algorithms that function by building global solutions in a bottom-up fashion from solutions to subproblems.

**dynamic programming invariant** The assumption, in a search, that if the optimal path through a graph happens to go through state $q$, then this optimal path must include the optimal path up to and including $q$.

**expectation maximization (EM)** Method for learning the parameters of model by attempting to maximize the probability of a set of training examples.

**final state** A state at which a finite-state automaton can legally conclude the processing of input.

**finite-state acceptor** A finite-state automaton in which the symbols on the arcs are letters in an alphabet and the automaton itself represents a set of legal strings.

**finite-state automaton (FSA)** A model, generally represented as a directed graph, made up of a finite number of states with arcs representing the transitions between states. Symbols on the arcs represent the conditions that must be met to transition between states.

**finite-state transducer** A finite-state automaton in which the symbols on the arcs are pairs of letters and the automaton itself represents a set of legal string pairs.

**forward algorithm** A dynamic programming algorithm which calculates the the sum of the probabilities of all paths through a statistical model that correspond to the input.

**forward transliteration** The transliteration of word from its native writing script to a foreign one.

**forward-backward algorithm** A dynamic programming approach to implementing EM learning. Stores the probabilities for alignments of prefixes and suffixes of a word pair and uses these probabilities to collect partial counts.

**fuzzy string matching** Similarity metric based on creating equivalence classes between letters in a pair of words.

**identity** Edit operation in which a character or group of characters in the source string maps to the same character or group of characters in the target string.

**insertion** Edit operation in which a character or group of characters in the target string are not mapped to anything in the source string.

**inversion** An operation on a finite-state transducer in which the left and right sides of the symbol pairs on the arcs are reversed.

**language model** The part of a noisy channel model that defines $P(E)$. Used to rate words on how well-formed they are.

**letter-based transliteration** Machine transliteration paradigm in which the basic unit of transliteration is the letter.

**Levenshtein edit distance** A similarity measure between two words. The distance is the count of insertions, deletions and substitutions required to convert the source string into the target string. Identities are not counted towards the distance score.

**logographic script** Script which uses symbols to represent morphemes in a given language.

**lossy** Refers to processes in which information is lost.

**machine transliteration** The task of automating the transliteration of words from one writing script to another.

**monotone search algorithm** A linear time decoding algorithm for phrase-based translation which assumes that the translation of phrases occurs sequentially.

**noisy channel model** Statistical approach to transliteration which uses Bayes' rule to split the probability $P(E|F)$ into $P(E)$ (the language model) and $P(F|E)$ (the transliteration model).

**null symbol** Represents an "empty" string or character. For example, a deletion can be described as mapping a letter in the source string to null.

**partial counts** Counts of mapping operations in training data weighted by the probabilities of the alignments in which they appear.

**phoneme-based transliteration** Machine transliteration paradigm in which the basic unit of transliteration is the phoneme.

**phrase-based translation** A framework for machine translation based on learning mappings between phrases in different languages rather than individual words.

**Romanization** An orthographic mapping from a word in a non-Latin writing script into Latin letters. Based on the spelling of the original word, rather than its pronunciation.

**shortest path search** see *Dijkstra's algorithm.*

**source string** In generative processes, the string that generates the target string. In joint processes, the term is simply used to differentiate it from the target string.

**start state** The state in which a finite-state automaton begins when processing input.

**statistical machine translation (SMT)** An approach to machine translation which models translation probabilistically.

**substitution** Edit operation in which a character or group of characters in the source string map to a character or group of characters in the target string.

**substring-based transducer** Substring-based transliteration model which encodes the transliteration process as the composition of two transducers: one to model mappings and the other to model language.

**substring-based transliteration** Machine transliteration paradigm in which the basic unit of transliteration is the word substring.

**syllabic script** Script which uses symbols to represent syllables in a given language.

**target string** In generative processes, the string that is being generated by the source string. In joint processes, the term is simply used to differentiate it from the source string.

**transliteration** The process of converting a word from one writing script to another, usually based on the phonetics of the original word. Transliteration is most often used for named entities, but is occasionally used for borrowed words as well.

**transliteration model** The part of a noisy channel model that defines $P(F|E)$. Used to rate pairs of strings based on how well they map to each other.

**true alphabet** Alphabet in which vowels and consonants are written as independent letters.

**union** An operation on a set of finite-state transducers which results in an FST that can accept all the same input as the original transducers and will produce all the same output.

**Viterbi algorithm** A dynamic programming decoding algorithm for finding the single most probable path through a model.

**Viterbi substring decoder** Substring-based transliteration model which uses dynamic programming to decode. Based on the monotone search algorithm for machine translation.

**weighted finite-state automaton (WFSA)** A finite-state automaton with weights on the arcs representing the probability of making a particular transition.

**weighted finite-state transducer (WFST)** A finite-state transducer with weights on the arcs representing the probability of making a particular transition.